# Enabling High-Performance and Secure Userspace NVM File Systems with the Trio Architecture

Diyu Zhou
EPFL

Vojtech Aschenbrenner
EPFL

Tao Lyu
EPFL

Jian Zhang
Rutgers University

Sudarsun Kannan
Rutgers University

Sanidhya Kashyap
EPFL

## Abstract

Userspace library file systems (LibFSes) promise to unleash the performance potential of non-volatile memory (NVM) by directly accessing it and enabling unprivileged applications to customize their LibFSes to their workloads. Unfortunately, such benefits pose a significant challenge to ensuring metadata integrity. Existing works either underutilize NVM's performance or forgo critical file system security guarantees.

We present Trio, a userspace NVM file system architecture that resolves this inherent tension with a clean decoupling among file system design, access control, and metadata integrity enforcement. Our key insight is that other state (*i.e.*, auxiliary state) in a file system can be regenerated from its "ground truth" state (*i.e.*, core state). Thus, Trio explicitly defines the data structure of a single core state and shares it as common knowledge among its LibFSes and the trusted entity. Enabled by this, a LibFS can directly access NVM without involving the trusted entity and can be customized with its private auxiliary state. The trusted entity enforces metadata integrity by verifying the core state of a file when its write access is transferred from one LibFS to another. We design a generic POSIX-like file system called ArckFS and two customized file systems based on the Trio architecture. Our evaluation shows that ArckFS outperforms existing NVM file systems by 3.1× to 17× on LevelDB while the customized file systems further outperform ArckFS by 1.3×.

***CCS Concepts*** • **Hardware** → **Non-volatile memory**; • **Information systems** → **Phase change memory**; • **Software and its engineering** → **File systems management**;

***Keywords*** Userspace File Systems, Library File Systems, Direct Access, File System Customization, File System Integrity, Persistent Memory

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

*SOSP '23, October 23–26, 2023, Koblenz, Germany*

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613171

**ACM Reference Format:**
Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. 2023. Enabling High-Performance and Secure Userspace NVM File Systems with the Trio Architecture. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3600006.3613171

## 1 Introduction

Emerging non-volatile memory (NVM) technologies, *e.g.*, Intel Optane persistent memory [11] and future CXL-based storage devices [14, 27], offer the best of memory and storage. NVM's unique characteristics provide new opportunities to design high-performance file systems. Specifically, NVM allows direct access with unprivileged load and store instructions. Furthermore, the hardware memory management unit (MMU) enforces access permission to different NVM regions, thus deprecating the need for a privileged entity to mediate every NVM access. These observations lead to userspace NVM file system designs that move (most) parts of the file system functionality out of the kernel to an application-linked library file system (LibFS) [20, 23, 32, 35, 38, 46].

Userspace NVM file systems bring two key performance advantages. First, applications can *directly access* NVM through LibFS to perform file system operations, thereby minimizing the software overhead in the storage stack. Second, applications can *customize* its LibFS to bridge the semantic gaps and further boost its performance [46]. Critically, an application does not require special privileges to customize its LibFS. Moreover, customizing for one workload may often negatively affect another. Userspace NVM file systems can effectively avoid this issue by assigning private LibFSes to each application.

Unfortunately, the performance advantages also pose an inherent challenge to enforcing file system *security*. Specifically, with *direct access*, malicious applications can bypass enforcement and attack others by corrupting file system metadata. Resolving this requires a trusted entity to validate metadata integrity. However, due to *customization*, each LibFS' data structures may be different, which the trusted entity does not understand, thus preventing the validation.

Existing designs fail to resolve such inherent tension. The conventional design [20, 32, 35, 46, 48] employs a trusted

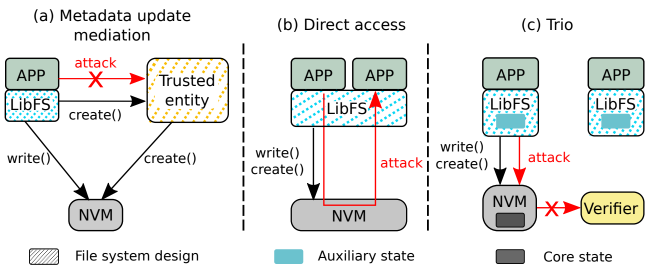

**Figure 1.** Comparison of userspace NVM file system architectures. (a) Designs [20, 32, 35, 46, 48] that mediate metadata updates prevent direct NVM access and hinder unprivileged private customization. (b) Designs [23, 38] that support direct NVM access incur security vulnerabilities since malicious applications can corrupt shared file system state. (c) Trio enables direct NVM access, unprivileged private customization, and metadata integrity through state separation and a clean decouple of responsibilities.

entity to mediate and perform metadata updates. While this design ensures metadata integrity, it incurs expensive overhead for metadata updates (up to 68%, as reported in [23]), bakes file system design into the trusted entity, and therefore requires special privileges for customization. It thus significantly compromises the aforementioned performance advantages. On the other hand, ZoFS [23] enables direct metadata updates but forgoes metadata integrity. ZoFS offers protection by confining metadata corruption within one set of NVM pages (*i.e.*, coffer). Despite this, since applications share coffers, malicious applications can still perform attacks by corrupting the metadata *within* a coffer (§2.3.2).

We present Trio, a new userspace NVM file system architecture that provides both performance and security by supporting three crucial properties simultaneously: (1) **Direct access:** LibFSes can directly access NVM for both data and metadata operations on both regular files and directories. (2) **Unprivileged private customization:** Applications can flexibly customize their LibFSes without special privileges (unprivileged) or affecting others (private). (3) **Metadata integrity:** As with kernel file systems, a malicious application cannot attack others by corrupting the metadata state.

To overcome the inherent tension among the design goals, our key insight is that file system state can be *separated* into core state and auxiliary state. Core state is the essential state that a file system uses as ground truth (*e.g.*, inode, data pages), while auxiliary state is decided by and can be rebuilt from core state (*e.g.*, in-memory cache, block bitmap). Using this insight, Trio explicitly defines the layout and data structure of a single core state, which are shared among components while each component maintains its private auxiliary state.

Trio consists of three (types of) components: an in-kernel access controller, per-application LibFSes, and a trusted userspace integrity verifier. The kernel controller decides which shared file system resources (i.e., NVM pages and inodes) a LibFS can access. Each LibFS realizes a complete file system design, *directly accesses* a file[1]'s core state for data and metadata operations, and performs *unprivileged private customization* with its private auxiliary state.

To ensure *secure sharing*, Trio prevents LibFSes from updating a file simultaneously. Furthermore, before a LibFS accesses a file modified by another LibFS, the integrity verifier checks if the file's core state is valid. If not, the kernel controller handles the corruption. Unlike a full offline file system check, the integrity verifier only checks the core state of a single file, making its overhead acceptable (ranges from several to hundreds of microseconds for medium-sized files, as reported in §6.5). The LibFS then rebuilds the file's auxiliary state from the valid core state and can securely access the file. Trio thus departs from the prior approaches that either validate metadata integrity for every operation [20, 32, 35, 46] or completely forgo it [23, 38].

We demonstrate the advantages of Trio by designing a generic POSIX-like (but not fully POSIX-compliant) file system: ArckFS[2]. To fully exploit the benefits of *direct access*, we carefully design ArckFS with efficient data structures, scalable NVM data access engine, and fine-grained parallelism, thereby achieving low latency, high bandwidth, and excellent scalability for both data and metadata operations.

Using Trio, we further design two customized file systems based on ArckFS's core state: KVFS [46], which optimizes for small file access, and FPFS [45, 53], which optimizes for deep directory access. The customization involves changing interfaces and key metadata structures and is heavily workload-specific. By making file system design solely the responsibility of LibFS, Trio enables the customization without special privilege or affecting other applications, neither of which prior approaches [46] could achieve. Our evaluation shows that ArckFS outperforms other file systems by by 3.1× to 17× on LevelDB, and has two orders of magnitude improvements in scalability microbenchmarks. KVFS and FPFS further outperform ArckFS by 1.3×.

This paper makes the following contributions:

- **Trio.** We propose a new userspace file system architecture: Trio. Through file system state separation and clean responsibility decoupling, Trio provides *direct access*, *unprivileged private customization*, and *metadata integrity* simultaneously.
- **ArckFS.** We design ArckFS, a POSIX-like userspace NVM file system based on Trio. ArckFS achieves low latency, high throughput, and excellent scalability for both data and metadata operations.
- **File system customization.** We present two customized file systems, demonstrating the flexible unprivileged private customization enabled by Trio.

[1]We use the term "file" to refer to both regular files and directories.
[2]named after Beethoven's Piano Trio "Archduke"

| | In-kernel | Metadata update mediation (§2.3.1) | Direct access (§2.3.2) | Trio |
|---|---|---|---|---|
| Direct data access | × | ✓ | ✓ | ✓ |
| Direct metadata access | × | × | ✓ | ✓ |
| Unprivileged custom. | × | × | ✓ | ✓ |
| Per-app custom. | × | × | × | ✓ |
| Metadata integrity | ✓ | ✓ | × | ✓ |

**Table 1.** Summary of NVM file systems.

## 2 Background and Motivation

Our work and many related NVM file systems are not limited to a specific NVM technology. This section presents our hardware assumptions on NVM (§2.1), a motivation for file system customization (§2.2), existing userspace NVM file systems designs (§2.3), and a summary of lessons we learned from prior designs (§2.4), which motivates Trio.

### 2.1 NVM Technologies

**Hardware assumptions.** Across the paper, we use the term "NVM" to denote storage devices with the following characteristics. First, software can access NVM through unprivileged instructions (e.g., load/store). Second, there exist mechanisms (e.g., page tables) for privileged software to enforce access permission to NVM regions. Third, the access latency of NVM is low. Fourth, NVM is byte addressable. The first two characteristics enable secure userspace access. The third characteristic motivates reducing software overhead, and the last characteristic is critical to the file system design.

**Present and future use of NVM.** Many storage devices meet the above assumptions. Specifically, industry has been using battery-backed DRAM as NVM [10, 34]. Intel Optane Persistent Memory [11], based on 3D Xpoint [1], is the first publicly available NVM technique. Emerging Compute Express Link (CXL) standard [3] introduces new possibilities to NVM. The current CXL standard includes specific support for NVM [2]. Industry has already proposed CXL-based NVM devices with new memory technologies [12, 27] or a combination of battery-based DRAM and flash memory [14].

### 2.2 File System Customization

A general-purpose system aims to provide acceptable performance under all scenarios and thus cannot provide optimal performance for a specific workload. As a result, improving application performance with workload-specific customization of the underlying system software has been extensively studied [25, 30, 33, 40, 41, 46, 52].

Prior works have built customization frameworks for scheduling [33], kernel locks [40], and the entire operating system [25, 30]. These frameworks aim to provide two key characteristics. First, customization can be performed securely without special privileges. Thus, all applications in the system can benefit from customization. Second, customization performed by one application does not affect other applications. This is critical since a custom mechanism that optimizes performance for one workload can reduce performance for others. However, as detailed below, existing userspace NVM file systems cannot fully achieve the above two characteristics.

### 2.3 Userspace NVM File Systems

Prior research on userspace NVM file systems has deeply investigated the direct access and customization benefits. They have proposed novel designs to enable direct access for data [32] or even metadata operations [23, 54]. In addition, `Aerie` [46] pioneers userspace NVM file system customization and proposes flexible interfaces to efficiently implement LibFSes. Existing works on userspace NVM file systems also explore leveraging NVM in tired storage [35], maximizing multicore scalability [20], or minimizing indexing overhead [38].

Userspace NVM file systems typically assume a threat model where hardware, kernel, and privileged userspace processes are not compromised and thus trusted. However, LibFSes and applications are not trusted and can cause arbitrary corruption in the file system. As a result, the design of userspace NVM file systems focuses on preventing metadata (*i.e.*, any state other than actual data in regular files) corruption from malicious applications and LibFSes. Despite their impressive achievements, existing designs either enforce metadata integrity at a high performance cost (§2.3.1) or relax the guarantee, leading to security vulnerabilities (§2.3.2).

#### 2.3.1 NVM FS based on metadata update mediation

One type of userspace file systems achieves metadata integrity by preventing LibFSes from directly updating it [20, 32, 35, 46]. Instead, a trusted entity, either in the kernel or as a privileged process, receives metadata update requests from LibFSes, validates the requests, and performs the update for the LibFS. This introduces an undesirable coupling between file system design and the trusted entity and thus significantly impacts performance, as we detail below.

**Increase software overhead for metadata updates.** All metadata updates involve expensive IPCs or context switches between the LibFS and the trusted entity, significantly increasing the software overhead. `ZoFS` [23] reports that such mediation incurs an overhead of 44% for 4KB appends and 68% for file creations. Furthermore, synchronizing metadata access between the trusted entity and LibFSes introduces a scalability bottleneck. Existing designs rely on complex lock-free algorithms to resolve it [20].

`Aerie` [46] and `Strata` [35] alleviate the communication overhead by batching the update requests in a local log. This incurs an extra write to the log. The log also requires garbage collection mechanisms, leading to additional overhead and complexity. Finally, a log shared by all threads may become a scalability bottleneck, while a per-thread log requires a complicated algorithm for correctness [18].

**Hinder customization.** With this design, file systems bake the implementation of metadata updates into the trusted entity. Thus, to customize, for example, a data structure used in metadata updates, an application needs special privileges to change the trusted entity. Furthermore, such changes affect all applications that share the trusted entity.

### 2.3.2 NVM FS based on direct metadata updates

Another type of userspace file systems relaxes the metadata integrity guarantee to enable direct metadata updates [23, 38, 54]. `ctFS` [38] does not enforce metadata integrity, assuming that all applications sharing the file system trust each other. `MadFS` [54] allows direct updates to certain metadata in regular files and thus similarly suffers from some of the vulnerabilities mentioned below.

**`ZoFS` overview.** `ZoFS` allows direct metadata updates and offers protections with the coffer abstraction [23]. A coffer contains multiple NVM pages with the same access permission. `ZoFS` couples a LibFS to a coffer instead of an application as in prior designs (§2.3.1). Hence, applications access a shared coffer through the same LibFS and share that LibFS's state (Figure 1). `ZoFS` does not prevent metadata corruption but instead mitigates its effects. For example, `ZoFS` uses Intel MPK [13] to make LibFSes access only one coffer at a time, thereby confining the metadata corruption within that coffer.

**`ZoFS` limitations.** Since multiple applications use the same LibFS to access one coffer, customizing one LibFS affects all the applications sharing that coffer.

`ZoFS` does not enforce metadata integrity and allows applications to update the shared coffer and LibFS state simultaneously. Although `ZoFS` can prevent some attacks, these two design choices lead to several vulnerabilities. Some examples include (1) Memory-based exploitation. A malicious application (attacker) can modify pointers in file system data structures, causing another application (victim) to leak or overwrite sensitive information in DRAM. For example, suppose a victim copies a file. The attacker can modify the pointers in the source file's indexes to point to the victim's sensitive DRAM data, making the victim write it to the destination file. (2) Denial of service attack. For example, as noted by `ZoFS`'s authors, an attacker can hold locks in the LibFS forever. (3) Semantic attack. An attacker can perform attacks by violating file system guarantees. For example, an attacker can remove non-empty directories, making files disconnected from the root path. Other examples include creating files with the same name under one directory or causing loops in directory paths.

## 2.4 Lessons Learned from Prior Designs

**Avoid metadata update mediation.** Mediating metadata updates incurs significant overhead, hinders customization, introduces scalability bottlenecks, and greatly complicates system design (§2.3.1).

**Enforce metadata integrity with trusted entities.** Not enforcing metadata integrity leads to severe security vulnerabilities (§2.3.2). While it is possible to let a LibFS check the metadata integrity before its access, this imposes a significant burden on programming LibFS. Furthermore, LibFS does not have the knowledge and the privilege to handle corruption (*e.g.*, revoke file system access permission from a malicious application). Thus, central trusted entities should enforce metadata integrity.

**No simultaneous state sharing among applications.** Sharing LibFS state among applications prevents per-application customization and incurs security vulnerabilities (§2.3.2). Thus, each LibFS should belong to only one application. Concurrent writes to NVM pages (*e.g.*, coffer) from multiple applications also open up opportunities for attacks. Thus, as prior designs (§2.3.1), only concurrent read access or exclusive write access to a file should be allowed.

## 3 The Trio Architecture

Motivated by §2, we present Trio, a userspace file system architecture that unleashes the performance potential of NVM while ensuring metadata integrity. This section presents Trio's design goals and challenges (§3.1), design overview (§3.2), and concludes with a discussion (§3.3).

### 3.1 Trio Design Goals and Challenges

We design Trio to meet the following goals and resolve the inherent tension among these goals.

**Direct access.** To minimize software overhead and avoid scalability bottlenecks, Trio allows a LibFS to access NVM directly to perform both data and metadata operations on both regular files and directories.

**Unprivileged private customization.** To facilitate *unprivileged* customization, Trio must cleanly decouple file system design from trusted entities; file system design is only LibFSes' responsibility. Furthermore, a LibFS should not be shared so that applications can customize their LibFSes without affecting others.

**File sharing.** To maintain the conventional file system abstraction, Trio should allow multiple applications, each has a different private LibFS, to share files. However, with *unprivileged private customization*, each LibFS has its own file system semantics. It is challenging to enable *file sharing* among LibFSes with, for example, different data structures.

**Metadata Integrity.** Since applications can be buggy or malicious, as with kernel file systems, Trio must prohibit ill-behaved applications from (1) accessing data without permission and (2) conducting attacks on others by corrupting the file system's metadata.

It is advantageous for trusted entities to enforce metadata integrity (§2.4). However, Trio's other design goals make it challenging. First, to meet the *direct access* design goal, trusted entities cannot use the conventional approach to mediate metadata updates. Second, due to file system customization, the trusted entities do not understand LibFSes' data structures and thus cannot verify their integrity.

## 3.2 Trio Overview

**File system state separation.** To overcome Trio's design challenges, we consider the state in a file system. We find that it can be separated into (1) "core state", which is the essential state of a file system and cannot be generated if lost (*i.e.*, hard state [42]), and (2) "auxiliary state", which can be discarded and if necessary, rebuilt from the core state (*i.e.*, soft state). Hence, Trio can use core state as common knowledge among components for sharing and integrity enforcement while achieving its performance goals with auxiliary state.

More specifically, core state contains the most important file system state: contents of files and the critical metadata to realize the basic file system abstraction (*e.g.*, file names, file access permissions, the directory hierarchy). Auxiliary state contains the information to maintain file system interfaces (*e.g.*, file descriptors), to achieve supporting functionalities (*e.g.*, locks), and to speed up accessing core state (*e.g.*, various types of caches, the inode bitmap). Core state must be in non-volatile storage since it cannot be lost, while auxiliary state can be in either non-volatile or volatile storage. The exact core state and auxiliary state depend on the file system design. We provide one example of POSIX-like file systems in §4.

To achieve the design goals, Trio first explicitly defines the data layout and the data structure of a single core state. All the components of Trio, namely all LibFSes and the trusted entities, share this core state. A LibFS cannot change the data structure of the core state. Instead, each LibFS manages and can freely change its private auxiliary state. As detailed below, such state separation enables Trio to overcome its design challenges and simultaneously meet all the design goals.

**Trio components.** Figure 2 shows the components of Trio and its workflow. Unlike prior architectures [20, 23, 32, 35, 46], Trio cleanly decouples file system design, access control, and integrity verification from each other. Specifically, Trio consists of three (types of) components: per-application LibFSes, an in-kernel access controller, and a trusted userspace integrity verifier as a standalone privileged process. Each LibFS has its own file system design based on its private auxiliary state. The kernel controller enforces each LibFS' access to the shared file system resources (*e.g.*, NVM pages, inodes). Upon sharing a file, the integrity verifier inspects the file's core state modified by a LibFS (or a malicious application) to enforce metadata integrity.

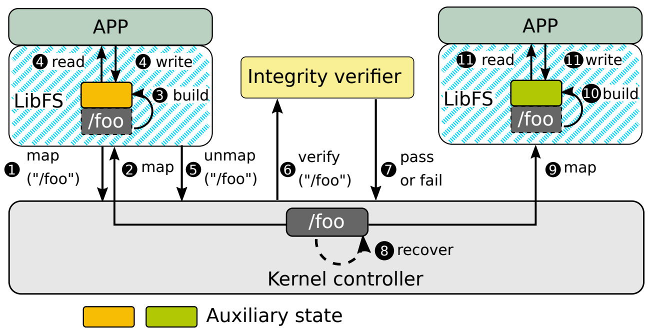

**Figure 2.** Trio enables file sharing between two different LibFSes. (1) A LibFS requests access to a file, and (2) the kernel controller grants it access to the file's core state. (3) The LibFS builds the file's auxiliary state. (4) Afterwards, the LibFS directly accesses the file. (5) Upon sharing, the LibFS unmaps the file, and (6) the kernel controller sends the file for verification. (8) If fails, the kernel controller handles the corruption (§4.3). (9) The kernel controller grants the other LibFS access to the file's valid core state. (10) The LibFS builds the file's auxiliary state and (11) accesses the file.

**Protected direct access.** With Trio, a file's NVM pages only contain the state of that file. When a LibFS accesses a file for the first time, it requests the kernel controller access to the core state of the file. If it has permission, the kernel controller grants access by programming the MMU. After obtaining access, a LibFS can access the core state for both metadata and data operations without involving any trusted entity, thereby achieving the *direct access* goal.

**Flexible unprivileged private customization.** Thanks to the state separation, file system design is solely the responsibility of LibFSes. Furthermore, each LibFS only belongs to one application. An application can thus freely perform customizations on the LibFSes' auxiliary state, thereby achieving the *unprivileged private customization* design goal (§5).

**File sharing among different LibFSes.** The sharing granularity in Trio is a single file. Since every LibFS understands the data structure of core state, despite having a different design, each LibFS can share a file by building its auxiliary state from the core state. As detailed below, Trio enforces a concurrent reads or exclusive write file sharing policy. Hence, a LibFS does not need to handle stale auxiliary state; either it is the only writer, or the file is read-only.

**Enforcing metadata integrity.** Enforcing *metadata integrity* requires avoiding simultaneous sharing of LibFSes and files state and preventing metadata corruption (§2.4). To avoid simultaneous state sharing, Trio associates a LibFS to one application and enforces concurrent read accesses or exclusive write access to a file.

Trio offers a guarantee that metadata corruption is confined to the application that causes it. As a result, Trio prevents attacks from malicious applications while enabling direct access. To realize this guarantee, when one LibFS releases its write access to a file, Trio informs the integrity verifier to check the file's core state. The LibFS then waits for the verification (and the potential fix for state corruption) to complete. If the verification fails, the integrity verifier informs the kernel controller to handle it. The kernel controller can handle the corruption with various policies, such as preventing future access to the file, rolling back the file to a checkpoint state, or fixing the state corruption. Section 4.3 details one checkpoint-based policy that also minimizes data loss by allowing LibFSes to *commit* changes.

**Controlling the trust boundary.** With Trio, sharing a file across the trust boundary incurs the overhead of file mapping and unmapping, integrity verification, and rebuilding the auxiliary state. Following the conventional OS design, the default trust boundary in Trio is a process. However, such a trust boundary is often too restrictive since processes may mutually trust each other (*e.g.*, processes belonging to the same application). Trio's design enables a user to control the trust boundary explicitly, thereby improving sharing performance. Specifically, Trio provides an abstraction named trust group, which contains multiple processes belonging to the same user and mutually trusting each other. Thus, processes in the same trust group can share files with a shared LibFS and thereby avoiding the sharing overhead.

### 3.3 Discussion and Limitations

**Data integrity.** In essence, ArckFS aims at offering the performance advantages of userspace file systems (by maximizing *direct access* and *unprivileged private customization*) while preserving the same abstraction (by enabling *file sharing*) and security guarantee (by enforcing *metadata integrity*) of kernel file systems. Therefore, enforcing data integrity (*i.e.*, protecting data in regular files) is orthogonal to Trio's design. Since Trio already enforces file access permissions (with MMU as discussed in §3.2), as with kernel file systems, the user is responsible for managing file access permissions to protect against malicious users. For unintentional corruptions caused by, *e.g.*, software bugs or hardware errors, file systems based on ArckFS can design its core state to employ the relevant protection techniques.

**Limitations.** Trio inherits two general limitations from prior userspace NVM file systems [20, 32, 35, 46]. First, file systems based on the Trio architecture cannot fully conform to certain file system semantics like POSIX. Specifically, to ensure metadata integrity, Trio cannot support concurrent write accesses from multiple untrusted applications to a file. Other examples include MMU cannot enforce the search permission of a directory, and file timestamps under certain scenarios cannot be precisely maintained (*e.g.*, the access time of a read-only file). Second, Trio incurs extra costs upon file sharing. However, the sharing cost is not incurred when the file is read shared, or shared within a process or a trust group. The sharing cost is high only when multiple untrusted applications frequently write to a shared file. File

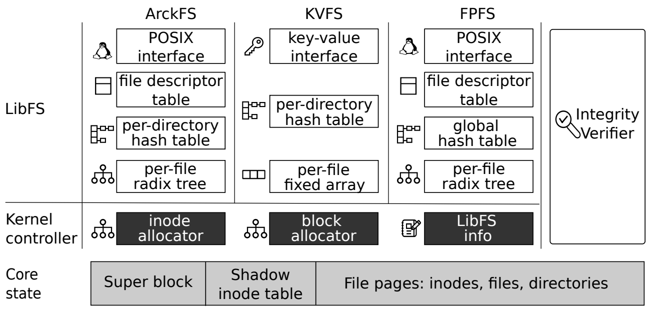

**Figure 3.** An overview of the three presented userspace NVM file systems using the Trio architecture. All the LibFSes share the same core state, kernel controller, and integrity verifier.

systems in a central trusted entity are more suitable for the above scenario.

## 4 ArckFS: POSIX-like FS Using Trio

This section concretizes the design of Trio by presenting ArckFS, a userspace file system using the Trio architecture. ArckFS provides the POSIX APIs with similar file system semantics. In addition to the properties enabled by Trio (§3.1), ArckFS further features: (1) *Minimal core state.* Deciding the core state of a file system involves tradeoffs. A large core state reduces the building time of the auxiliary state upon sharing and simplifies LibFS development. A small core state increases the flexibility of customization and reduces the time in verifying metadata integrity. Thus, ArckFS chooses a minimal core state design. (2) *Multicore scalability.* Modern servers have hundreds of CPUs. While NVM allows a high degree of concurrent access, the storage stack often introduces scalability bottlenecks (*e.g.*, VFS [20, 39]). ArckFS maximizes concurrent access with fine-grained parallelism.

The rest of the section presents ArckFS 's core state (§4.1), how it achieves high performance and multicore scalability(§4.2), enforces metadata integrity(§4.3), and preserves crash consistency (§4.4).

### 4.1 Core state

**Layout.** As shown in Figure 3, ArckFS's core state consists of a superblock, a shadow inode table, and file pages. The superblock records the general file system information (*e.g.*, the total number of blocks). The kernel controller maintains the shadow inode table to enforce metadata integrity (§4.2). As detailed below, the file pages contain the inodes and data of directories and regular files. A LibFS has read access to the superblock, has no access to the shadow inode table, and the file access permission decides a LibFS' access to file pages.

Next, we present the data structure design in file pages to minimize the core state while maximizing direct accesses by enabling MMU to correctly enforce the access permission for common file system operations.

**Core state of a regular file.** Common operations on a regular file are `read`, `write`, and `truncate`. `Read` requires file

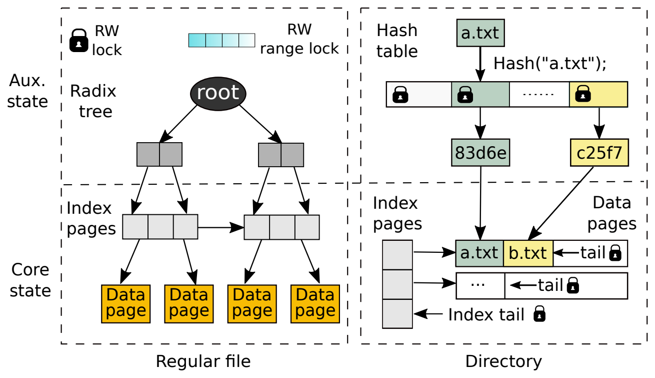

**Figure 4.** ArckFS's regular file and directory data structures (§4.2).

read permission, while `write` and `truncate` require write permission. Thus, to support direct userspace handling, a regular file contains index pages and data pages (Figure 4). Each entry of index pages points to a data page. The last entry of an index page points to the next index page. By mapping the index pages and data pages, the LibFS can directly handle `read`, `write`, and `truncate`. The LibFS persists data operations immediately (§4.4) and thus ignores `fsync()`.

**Core state of a directory.** Common operations on a directory are `search` a file under it, `list` all files under it, `stat` (obtain the statistics of) a file, `insert` a file, and `delete` a file. `Search`, `list`, and `stat` operations require directory read permission and `insert` and `delete` require write permission.

A directory contains index pages and data pages with directory entries. A directory entry contains information like inode number, file name, and the length of name. Thus, the LibFS can directly perform `list` and `search` by mapping the index and data pages of directories. As a result, LibFS resolves a path by mapping each directory's pages along the path and search for the next file.

As discussed above, `stat`, `create`, and `delete` require the read or write permission of the file's parent directory (instead of the file itself). To support direct userspace handling of these operations through page mapping, ArckFS co-locates a file's inode together with its directory entry. For example, in Figure 4, the "a.txt" (and "b.txt") block contains both the file's inode and its directory entry. Due to the co-location, there can be no "." and ".." in the core state of a directory. Instead, a LibFS maintains them in its auxiliary state. Section 4.3 discusses how to preserve the integrity of sensitive information in inodes (*i.e.*, file access permission).

## 4.2 Handling File System Operations with LibFS

ArckFS' LibFS handles file system operations by directly accessing its auxiliary state and the mapped core state (*i.e.*, files' index and data pages). In addition, occasionally, the LibFS issues request to the kernel controller to, *e.g.*, map or unmap files and allocate or free NVM pages. We next discuss the design of LibFSes' auxiliary state (as shown in Figure 4) to enable efficient and scalable file system operations.

**Regular file operations.** For a regular file, the LibFS uses a radix tree to map the offset within a file to the corresponding index page. Inspired by prior works [20, 43, 55], LibFS enables fine-grained concurrent access to a file with a readers-writer inode lock and a readers-writer range lock. Thus, within a process, LibFS allows one thread to append or truncate the file and multiple threads to concurrently write the disjoint region of the file and concurrent read from the file.

**Directory operations.** For each directory, the LibFS uses a resizable chained hash table to map the name of a file to its directory entry. In addition, the LibFS extends the per-inode log-structured design in `NOVA` [49, 50] by maintaining a logging tail for each non-full data page instead of a single logging tail in `NOVA`. Thus, threads can operate in parallel on different logging tails. To handle size increases, the LibFS also maintains the tail of the index pages (index tail). Each directory has three types of locks: per-bucket readers-writer locks in the hash table, a lock for each logging tail, and a lock for the index tail. As shown in §6.4, such a design achieves much better scalability than prior works.

**Building auxiliary state from core state.** For a regular file, the LibFS initializes a radix tree and iterates through the index pages to insert its address into the radix tree. It finishes the building by initializing the inode lock and the range lock. For a directory, the LibFS iterates the directory entries and inserts them into the hash table. It finishes the building by initializing the logging tails, the index tail, and their locks. A LibFS can preserve the auxiliary state of a file until another application requests to write to the file.

## 4.3 Enforcing Metadata Integrity upon Sharing

Trio enables confining metadata corruption within the application that causes it (§3.2). This subsection presents how ArckFS enforces this guarantee by detecting and fixing metadata corruption in a file's core state upon sharing.

**Detecting metadata corruption.** The integrity verifier in ArckFS is a trusted standalone process responsible for detecting metadata corruption. We refer to the integrity checks in the `ext4` file system checker (*i.e.*, e2fsck) [9, 17] and adapt them to ArckFS. We find that ArckFS' minimal core state greatly simplifies the integrity check. Also, the checks in ArckFS are quite different from e2fsck and include a few extra checks due to (1) the differences in file system data structures and (2) that e2fsck checks file system integrity globally and offline while ArckFS' integrity verifier operates on an individual file and online. The integrity verifier detects metadata corruption by checking the following invariances.

**I1: Fields in each inode and directory entry are valid.** The integrity verifier checks for invalid values of each field and inconsistency among fields within the file. Examples include (1) the file type is valid: either a directory or a regular file, (2) no file shares the same name under one directory, and (3) no "/" in file names.

**I2: A file's inode number, index pages and data pages are valid.** Specifically, a file's inode, index page, and data pages must either already exist before mapping to the LibFS or be allocated to the LibFS by the kernel controller. Furthermore, the inode and pages are not doubly referenced.

To perform these checks, the kernel controller maintains the following global file system information: (1) all the inodes and pages that are write-mapped or allocated to each LibFS and (2) all the inodes and pages that are in the existing files. The kernel controller updates such information when a LibFS maps or unmaps a file and allocates or frees inodes or data pages. The integrity verifier has read access to such information and can thereby perform the checks.

**I3: The directory hierarchy forms a connected tree.** Since ArckFS's directory core state does not contain "." and ".." (§4.1), the integrity verifier is freed from checking the inconsistency caused by them. Since ArckFS currently does not support hard links, enforcing *I2* already prohibits cycles in the directory hierarchy. Thus, enforcing *I3* means the integrity verifier only needs to ensure the directory tree is connected. To achieve this, the integrity verifier compares the directory under check against its checkpoint state (used for recovery as detailed below) to identify *deleted* child directories. The integrity verifier then checks that the deleted child directory is not mapped to any LibFS and has no file under it.

**I4: The access permission is correctly enforced.** ArckFS uses MMU to enforce the access permission under most scenarios, with one case requiring extra handling. Specifically, inodes mapped to LibFSes contain access permissions (§4.1). Therefore, any application with write access to the directory can modify the permissions, leading to security vulnerabilities. To resolve this, the kernel controller maintains file access permission in the shadow inode table (§4.1) and uses them as ground truth; the ones in a normal inode are treated as cached state. To execute permission change operations (*i.e.*, `chmod` and `chown`), a LibFS issues request to the kernel controller to modify the access permission in the shadow inode.

**Fixing metadata corruption.** ArckFS handles corruption by rolling back the file's metadata to a checkpoint state while also including mechanisms to minimize data loss. Specifically, before the kernel controller grants one LibFS (LibFS A) write access to a file, it checkpoints the file's metadata (*i.e.*, index pages for a regular file; both index and data pages for a directory). Afterward, upon file sharing, if the verification fails, ArckFS notifies LibFS A to fix the corruption with a timeout. If LibFS A cannot fix the corruption, the kernel controller makes the corrupted file a private file to LibFS A and preserves LibFS A's access to avoid data loss.

The kernel controller then fixes the corruption by making a copy of the corrupted file and reverts the file's metadata state to a checkpoint state. An inconsistency might occur between the current and the checkpointed file size. The kernel controller resolves it by trimming or padding zero bits to the file. To further avoid data loss, the kernel controller provides a `commit` call for LibFSes to replace a file's checkpoint with the current state, given that the current state passes the integrity check, thereby ensuring the kernel controller will not revert the changes.

## 4.4 Crash Consistency

**Overview.** ArckFS' core state (§4.1) does not specify a crash consistency mechanism to enable LibFSes to design their own. Thus, when a LibFS registers with the kernel controller, it also needs to specify a program to handle crashes. Upon reboot after a crash, the kernel controller invokes this program to perform recovery. Since ArckFS cannot trust the programs provided by LibFSes, it invokes the integrity verifier after recovery to ensure the valid state of all files that are mapped as writable when the crash occurs.

**Consistency mode of the LibFS.** We next discuss the design of the crash consistency mechanism of ArckFS' LibFS. Similar to other designs [24, 31, 32, 49, 55], the LibFS ensures all metadata operations are synchronous (*i.e.*, the operation persists on NVM before the system call returns) and atomic (*i.e.*, no partial update). Data operations are synchronous but not atomic (*i.e.*, a partial write is possible upon crash). Extending the LibFS to support other consistency modes is simple by following the prior approaches [32, 49].

**Consistency mechanism in the LibFS.** Current hardware supports atomic NVM updates up to 16 bytes [24, 55]. For most operations, ArckFS's core state design (§4.1) enables LibFSes to use atomic updates to ensure crash consistency. For example, during file creation, LibFS first persists all other writes of the directory entry with an inode number 0 to mark the directory as invalid. Finally, it persists the update to the inode number. A few complex operations, such as `rename`, require journaling. ArckFS uses undo logs for simplicity.

## 4.5 Implementation

We implement the kernel controller as a module for the 5.13.13 Linux kernel. As with prior works [35, 46], the kernel controller uses leases to prevent a LibFS from holding a file forever. The LibFS' index structures, specifically the radix tree and the hash table (§4.2), are inspired by `ScaleFS` [18]. As with `NOVA` [49] and `WineFS` [31], the kernel controller and LibFS implement the heap and inode allocators in DRAM as red-black trees. However, since the aforementioned file systems are in the kernel, most of the time, we need to reimplement the data structures in the LibFS. We implement the kernel controller and the integrity verifier from scratch. The kernel controller, LibFS, and the integrity verifier have 3791, 7586, and 457 lines of code, respectively. The code of the integrity verifier is small since there is no complex data structure in ArckFS's core state, and verifying metadata integrity in runtime avoids many checks [26].

**Multicore scalability.** ArckFS's design enables fine-grained parallelism to directories and files (§4.2). In addition, we make key data structures in the kernel controller and LibFS per-CPU, including the block allocators, inode allocators, file descriptor allocators, and journal. ArckFS implements its readers-writer locks based on state-of-the-art synchronization techniques [22].

**Adapting to Intel Optane PM.** We implement ArckFS for the Intel Optane PM since it is the only publicly available NVM. Due to the hardware design of Optane PM, excessive concurrent access from multiple CPUs and remote NUMA access significantly degrades performance [21, 29, 47, 51]. Thus, ArckFS employs the opportunistic delegation technique in `OdinFS` [55] to maximize PM performance. Specifically, ArckFS creates multiple background kernel threads (delegation threads) in each NUMA node. The delegation threads are shared by all LibFSes. Application threads cannot access NVM but instead sends the access request to one of the delegation threads in the corresponding NUMA node with a per-application ring buffer. Afterward, the application threads wait for the delegation threads to perform and complete the access. The fixed number of delegation threads avoids performance collapse due to concurrent access. Moreover, delegation threads always perform local NVM access. Therefore, it scales NVM performance. Furthermore, by striping the file data across NVM in multiple NUMA nodes, ArckFS handles bulk data operations by letting delegation threads access NVM nodes in parallel, fully utilizing aggregated NVM bandwidth. Due to the communication overhead, ArckFS does not delegate small NVM access (read access less than 32KB and write access less than 256 bytes).

## 5 File System Customization

This section showcases Trio's unprivileged private customization advantage with two customized LibFSes: KVFS, which is very similar to the customization case in `Aerie` [46], and FPFS, which is based on full path indexing [45, 53]. KVFS and FPFS's designs are based on ArckFS's core state (§4.1), involving heavy changes in the interfaces and metadata structures of ArckFS's LibFS. Furthermore, KVFS and FPFS assume specific workloads and cannot (efficiently) handle other workloads or even certain basic file system operations. For example, FPFS cannot efficiently handle `rename`. Nonetheless, realizing KVFS and FPFS with Trio does not require modifying the trusted entities, and Trio enables deploying them to only applications that can benefit without affecting other applications. Neither of these two advantages is possible with `Aerie`.

**KVFS.** Applications, such as email clients [5] and some HPC applications [15, 16], operate on many small files. A generic file system incurs the overhead of file descriptors with the `open` and `close` interfaces. Furthermore, the file system also suffers from the overhead of managing and walking index structures. Both overheads are too high for small files.

We design KVFS to optimize applications working with many small files. Specifically, we add to ArckFS' LibFS `get` and `set` interfaces, that directly read from, or create and write to a specified file, respectively. The `get` and `set` APIs always operate from the beginning of a file. Thus, it does not need to maintain file descriptors, thereby eliminating their overhead. KVFS assumes a small maximal file size (32KB in our design). Thus, we replace the radix tree in ArckFS' auxiliary state with a fix-sized array, thereby minimizing the overhead associated with index structures. Finally, with many files, concurrent accesses from threads to the same file becomes unlikely. Therefore, we replace the fine-grained locks in ArckFS' auxiliary state with a simple spinlock to optimize for non-contended cases.

**FPFS.** With a typical file system, resolving a path requires iterating through each directory along the path. This approach incurs high overhead with deep directory hierarchies. We design FPFS for applications working with deep directory hierarchies. FPFS replaces the per-directory hash table in ArckFS' auxiliary state with a global hash table that directly maps a path to the corresponding directory entry in the core state. It thus eliminates directory traversals and significantly improves the performance of applications working with deep directory hierarchies.

**Customization limits.** To enforce metadata integrity, Trio limits the scope of customization to the auxiliary state in each LibFS (§3.2). Hence, customizations involving changing the core state require special privileges. For example, ArckFS's core state cannot support conventional log-structured file systems [37, 44]. Nonetheless, ArckFS's minimal core state (§4.1) can enable various customizations, including file system interfaces, caching mechanisms, key data structures, concurrency control, and crash consistency, which is more flexible than prior approaches like `Aerie`.

## 6 Evaluation

Our evaluation answers the following questions:

- What is ArckFS's performance with a single thread (§6.2) and multiple threads (§6.3)?
- Does ArckFS scale file system operations? (§6.4)
- Can ArckFS ensure metadata integrity upon malicious LibFSes and what is the sharing cost? (§6.5)
- How do ArckFS and the customized file system perform with macro-benchmarks and LevelDB? (§6.6)

### 6.1 Evaluation setup

**Environment.** Our evaluation machine has eight sockets and equips with 224-core Intel Xeon Platinum 8276L processors and 6144GB Optane PM DIMMs. The system runs Ubuntu 20.04 and Linux kernel 5.13.13.

**Baseline file systems.** We compare ArckFS against five in-kernel file systems: `ext4` [6], `PMFS` [24], `NOVA` [49], `OdinFS` [55], `WineFS` [31], and two userspace file systems: `SplitFS` [32], `Strata` [35]. `ext4` is a mature and widely used real-world file system and others are state-of-the-art NVM research file systems. `PMFS` is developed by Intel to exploit NVM characteristics, particularly the byte addressability. `NOVA` and `WineFS` improve upon `PMFS` with efficient data structures and faster crash consistency mechanisms (per-inode log in `NOVA` and per-CPU journaling in `WineFS`). `OdinFS` builds upon NOVA and WineFS with opportunistic delegation to maximize NVM performance (§4.5).

`SplitFS` and `Strata` are the only userspace file systems we can run and evaluate (as detailed below). Both handle data operations in userspace and require a trusted entity to handle metadata operations (§2.3.1). In the current implementation, `SplitFS` uses `ext4`, while `Strata` uses a privileged process.

**Configuration.** We configure the baseline file systems with the default setup except that we enable the DAX option for `ext4`. They provide weaker or the same crash consistency guarantee as ArckFS (§4.4). We evaluate ArckFS with two setups. The default one utilizes all eight NVM NUMA nodes, each with twelve delegation threads (following `OdinFS`' default setup). This setup demonstrates the maximal throughput and scalability ArckFS can achieve. For comparison, we configure `OdinFS` with the same setup. We further create a RAID0 of NVM nodes [4] and mount `ext4` on top of it (`ext4(RAID0)`). [3] To compare ArckFS against file systems that do not considerthe Optane characteristics discussed in §4.5, we also evaluate ArckFS on a single NUMA node without opportunistic delegation (ArckFS-no-dele).

**Workload.** Our workloads cover a wide range of file system use cases. For microbenchmarks, we use the popular `fio` [8] and FxMark [39] to evaluate latency, throughput, and multicore scalability, focusing on both data and metadata operations. We configure `fio` to let each thread access a 1GB private file. We use Webserver, Fileserver, Webproxy, and Varmail in Filebench [7] and LevelDB as macrobenchmarks.

**Limitations.** We test ArckFS' crash consistency with unit tests during development but do not evaluate it with testing frameworks such as Chipmunk [36]. We cannot run `Strata` beyond one thread (and thus only show its results in §6.2). We do not include `ZoFS` [23] and `KucoFS` [20] due to no source code access. Despite our best efforts, we are unable to run `Aerie` [46] and `ctFS` [38] with the provided configurations.

## 6.2 Single thread performance

Figure 5 shows the performance of common file system operations with a single thread. Due to space limitations, we present `NOVA` and `SplitFS` for data operations, representing the the best-performant kernel and userspace file systems, respectively. Similarly, we present `NOVA` and `Strata`

[3]Other evaluated NVM file systems cannot operate on a Linux RAID.

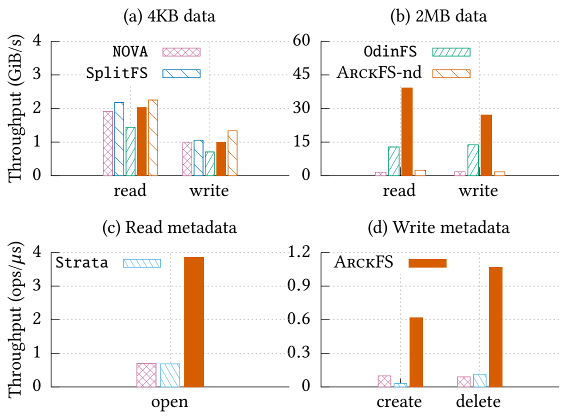

**Figure 5.** Single thread performance of the evaluated file systems.

for metadata operations. We present `OdinFS`' data operation results to show the advantages of direct NVM access in ArckFS. Since small NVM accesses are not delegated (§4.5), ArckFS-no-dele performs similarly to ArckFS for metadata operations and is thus omitted.

With the 4KB access size, `SplitFS` and ArckFS-no-dele outperform `NOVA` by 9% – 31% due to direct NVM access. The performance difference between `SplitFS` and ArckFS-no-dele is due to different implementations of `memcpy`. Opportunistic delegation introduces the overhead of striping data across NVM nodes and the communication overhead. As a result, ArckFS is 21% slower than ArckFS-no-dele but still outperforms `NOVA` by around 6%.

With the 2MB access size, the advantage of direct NVM access is minimal since the data copy time dominates. `OdinFS` and ArckFS parallelize data access with opportunistic delegation and thus perform much better than others. In this case, since application threads only need to send requests to delegation threads (instead of performing data copy), avoiding kernel trapping brings a significant advantage. In summary, ArckFS outperforms others by 3.1× to 25× and 2.0× to 15× for `2M-read` and `2M-write`, respectively.

Workloads for metadata performance include `open` and close a file in a five-depth directory, `create` an empty file, `delete` all the empty files under one directory. ArckFS's performance advantage comes from both direct NVM access and efficient data structures (§4.2). Taking `create` as an example, we find `NOVA` and `Strata` spend at least 42% and 44.5% of the time in VFS and digestion, respectively. Besides that, the rest of the performance difference is due to the data structure design. For example, with our workload, we find the index data structures in `NOVA` (radix tree) are slower than ArckFS's hash table. As a result, for `open`, `create`, `delete` ArckFS outperforms others by 1.6× to 5.6×, 3.3× to 5.3×, and 7.4× to 9.4×, respectively.

**Summary.** ArckFS outperforms due to direct access (for both data and metadata operations), opportunistic delegation (for data operations), and efficient data structures (for metadata operations).

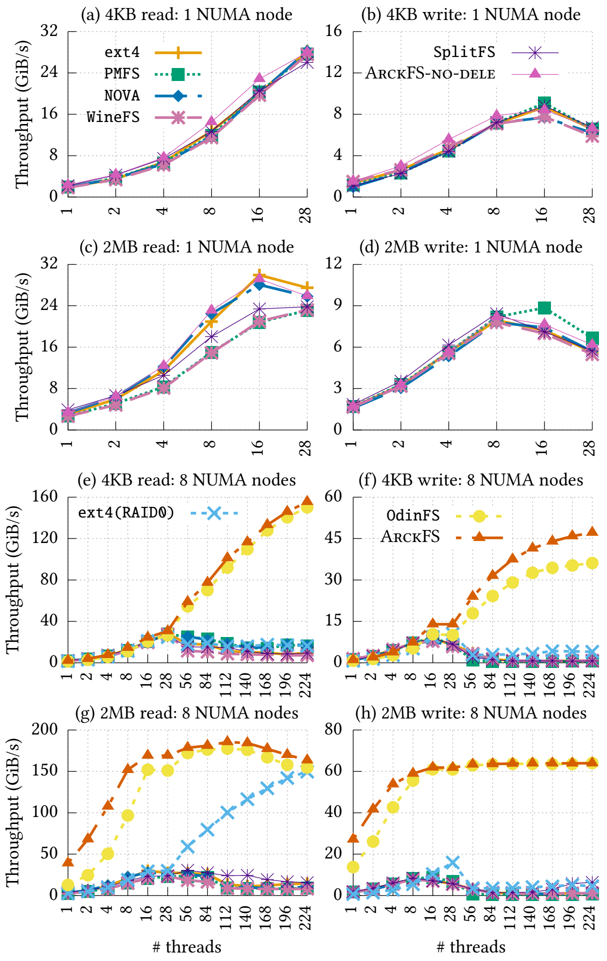

**Figure 6.** Throughput of the evaluated file systems with one and eight NUMA nodes. ArckFS follows `OdinFS`'s design to scale NVM performance and outperforms `OdinFS` with direct NVM access.

## 6.3 Data operation performance

Figure 6 shows the throughput of each file system with one and eight NUMA nodes evaluated with `fio`. With one NUMA node, for 4KB read and write, ArckFS-no-dele outperforms other file systems by 10% – 12% due to direct access. For 2MB read and write, all the evaluated file systems perform similarly. The throughput drop is due to excessive concurrent accesses (§4.5).

With eight NUMA nodes, for `4KB-read` and `4KB-write`, when the thread count is low, ArckFS (and `OdinFS`) performs similarly or worse than other file systems (§6.2). However, `OdinFS` and ArckFS can scale to 224 threads because they employ opportunistic delegation to preserve NVM performance (§4.5). ArckFS further outperforms `OdinFS` due to direct NVM access. With 224 threads, ArckFS outperform `OdinFS` by up to 1.3× and other file systems by up to 22×.

For `2MB-read` and `2MB-write`, since `OdinFS` and ArckFS parallelize NVM access (§6.2), they constantly outperform other file systems. `ext4(RAID0)` also scales `2MB-read` because

| Name | Description |
|---|---|
| `DWTL` | Reduces the size of a private by 4K. |
| `MRP(L/M/H)` | Open a (private/random/same) file in five-depth dirs. |
| `MRD(L/M)` | Enumerate files of a (private/shared) directory. |
| `MWC(L/M)` | Create an empty file in a (privte/shared) dir. |
| `MWU(L/M)` | Unlink an empty file in a (private/shared) dir. |
| `MWRL` | Rename a private file in a private dir. |
| `MWRM` | Move a private file to a shared dir. |

**Table 2.** Summary of FxMark's metadata microbenchmarks.

it reduces the degree of concurrent access to NVM and remote NVM reads incurs lower overhead than writes ([55]). `ext4(RAID0)` does not scale `4KB-read` due to a scalability bottleneck. When the thread count increases, the NVM bandwidth becomes the performance bottleneck and `OdinFS` starts to catch up with ArckFS. With 224 threads, ArckFS outperforms evaluated file systems by 1.1× to 25×, and up to 15× for `2MB-read` and `2MB-write`, respectively.

**Summary.** ArckFS follows `OdinFS`'s datapath design to scale NVM performance for both read and write operations. ArckFS further outperforms `OdinFS` with direct NVM access.

## 6.4 Scalability

We evaluate file system scalability (*i.e.*, performance with increasing number of threads) using the FxMark benchmark suite. Each benchmark in FxMark creates multiple threads and each thread repeats the same operation. Table 2 summarizes FxMark's metadata benchmarks.

**Data operation scalability.** Due to space limitations, we omit the figure of FxMark's data microbenchmarks. Except ArckFS and `OdinFS`, only `PMFS` and `NOVA` scale one workload: `DRBL`. ArckFS and `OdinFS` scale due to opportunistic delegation, fine-grained file access (§4.2), per-CPU data structures, and advanced lock design (§4.5). ArckFS further outperforms `OdinFS` with direct NVM access. As a result, ArckFS scales *linearly* in all the read-dominated workloads and maintains the maximal throughput in all the write-dominated workloads, up to 224 threads. In summary, ArckFS outperforms `OdinFS` by 3.2× and others by up to 850×, respectively.

**Metadata operation scalability.** Figure 7 presents the scalability of metadata operations. Essentially, for all the other evaluated file systems, VFS decides their scalability. Specifically, most other file systems can only scale `MRPL` and `MRDL` since the scalability bottlenecks in VFS (*e.g.*, coarse locks on directory cache, inode cache, directory inode and the global rename lock) prevent scaling other microbenchmarks [39].

Direct NVM access allows ArckFS to avoid the scalability bottleneck in the VFS. Furthermore, ArckFS scales due to its scalable data structures and lock design (§4.2, §4.5). Hence, for `DWTL` and all the read-dominated workloads, ArckFS scales linearly. Specifically, for microbenchmarks performing open and enumerate, at 224 threads, ArckFS outperforms others by 5.4× to 334× and 7.4× to 25×, respectively.

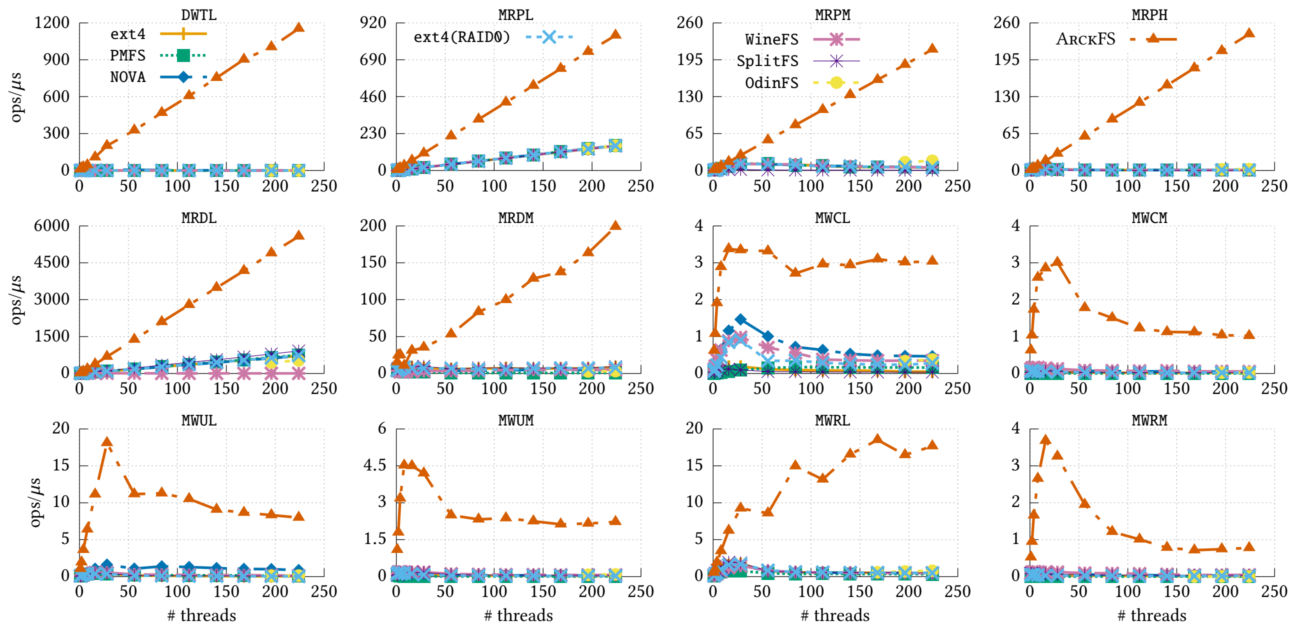

**Figure 7.** Metadata scalability of the evaluated file systems. ArckFS scales thanks to kernel bypassing and scalable design. Results with one NUMA node are similar to those under 28 threads and thus omitted.

For MWCL and MWUL, ArckFS does not scale linearly due to excessive concurrent NVM access; these small accesses are not delegated (§4.5). This does not affect MWRL since ArckFS writes much less NVM data for it. The scalability of MWCM, MWUM, and MWRM drops due to the contention in ArckFS's directory hash tables. Furthermore, MWCM and MWRM also contend on logging tails and the index tail (§4.2). In summary, for microbenchmarks that perform create, unlink, and rename, at 224 threads, ArckFS outperforms by 2.3× to 21.2×, 8.9× to 32.7×, 16.2× to 36.4×, respectively.

**Summary.** Direct NVM access allows ArckFS to bypass the scalability bottleneck in VFS. Leveraging this, the careful data structure design in ArckFS makes it scale significantly better than existing NVM file systems.

## 6.5 Metadata Integrity and Sharing Cost

**Detecting and recovering from metadata corruption.** We design tests emulating both malicious LibFSes and buggy LibFSes to stress the metadata integrity enforcement design in ArckFS (§4.3). Specifically, we handcrafted eleven attacks performed by a malicious LibFS corrupting metadata, some mentioned in §2.3.2. For example, the malicious LibFS (1) modifies pointers in index pages to point to DRAM data; (2) removes a non-empty directory; (3) creates file names containing "/" to trick another LibFS into accessing a wrong file; (4) causes loops within a file's index pages.

To emulate a buggy LibFS, for each integrity checks in the verifier, we create an automated script to corrupt the relevant metadata with, say, a random value. We also run different scripts together to cause more complex corruption. In total, we cause 134 corruption scenarios.

|  | NOVA | ArckFS | ArckFS-trust-group |
|---|---|---|---|
| 4KB-write 2MB | 1.92GiB/s | 1.90GiB/s | 1.95GiB/s |
| 4KB-write 1GB | 1.91GiB/s | 0.25GiB/s | 1.95GiB/s |
| Create 10 | 8.2$\mu$s | 7.8$\mu$s | 1.5$\mu$s |
| Create 100 | 8.4$\mu$s | 32.7$\mu$s | 1.6$\mu$s |

**Table 3.** Performance of ArckFS when two threads concurrently update the same file.

In all the test cases, the integrity verifier can detect the corruption, and the kernel controller can restore the corrupted file to a consistent state.

**Sharing cost.** When multiple untrusted applications concurrently update a file, ArckFS incurs a sharing cost caused by file mapping and unmapping, integrity verification, and rebuilding the auxiliary state. We evaluate the sharing cost with (1) two applications writing 4KB to a 2MB (`4KB-write 2MB`) or a 1GB file (`4KB-write 1GB`) and (2) two applications creating empty files in a directory that contains 10 (`create-10`) or 100 files (`create-100`).

Table 3 shows the results. When the file/directory size is small, the sharing cost is negligible ($< 5\mu$s) for `write` and modest for `create`, leading to similar performance as NOVA. The overhead increases when the file/directory size grows. Figure 8 shows the breakdown of the overhead. Specifically, for `4KB-write 1GB`, mapping and unmapping files, where each iteration takes 670ms on average, contribute to 99% of the overhead. With ArckFS's 100ms lease time, this results in an overhead of 7.8×. For `create`, we stress

| Name | # Files | Avg. file size | I/O size (r/w) | R/W |
|---|---|---|---|---|
| Fileserver | 10K | 2MB | 1MB / 512KB | 1:2 |
| Webserver | 20K | 4MB | 1MB / 256KB | 10:1 |
| Webproxy | 100K | 512MB | 1MB / 16KB | 5:1 |
| Varmail | 100K | 16KB | 1MB / 16KB | 1:1 |

**Table 4.** Filebench workloads configurations, which aim to cover a wide range of file system use cases.

the sharing overhead by making applications unmap a file after each operation. Thus, in this case, the overhead of verification (300$\mu s$, 81%), and rebuilding the auxiliary state dominates (12%), leading to an overhead of 20.4×. Leveraging the trust group (§3.2) can eliminate the overhead.

**Summary.** ArckFS can effectively enforce metadata integrity in the presence of buggy or malicious LibFSes. Concurrent write access to a shared file from untrusted applications incur sharing cost for ArckFS. If applicable, a user can use the trust group to avoid it.

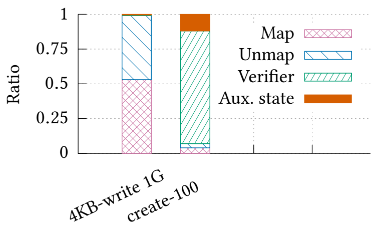

**Figure 8.** Breakdown of ArckFS' sharing cost.

## 6.6 Macrobenchmarks and Real-World Applications

**Filebench.** We use four Filebench benchmarks: Fileserver, Webserver, Webproxy, and Varmail with configurations shown in Table 4. We aim to cover a wide range of file system use cases with such configurations. Specifically, Fileserver and Webserver are data-intensive, performing large file writes and reads, respectively. Webproxy stresses both data and metadata operations, performing small file reads. Varmail is metadata-intensive, performing small file writes. We find that Filebench introduces a severe scalability bottleneck (by locking the whole fileset to choose a file for operations like open) for Webproxy and Varmail. We bypass it by assigning a private fileset to each thread. However, we cannot increase the fileset count beyond sixteen due to a bug in Filebench. Due to time limitations, we evaluate Webproxy and Varmail with only up to sixteen threads.

Figure 9 shows the result. For Fileserver and Webserver, with one NUMA node, all the file systems perform similarly. With eight NUMA nodes, ArckFS outperforms others by 1.1× to 27.3×, and 1.2× to 4.1×, respectively. These results are consistent with results in §6.3. Again, ArckFS and OdinFS outperform thanks to their datapath design that maximizes NVM performance, and ArckFS outperforms OdinFS due to direct NVM access and efficient metadata path (especially in Websever).

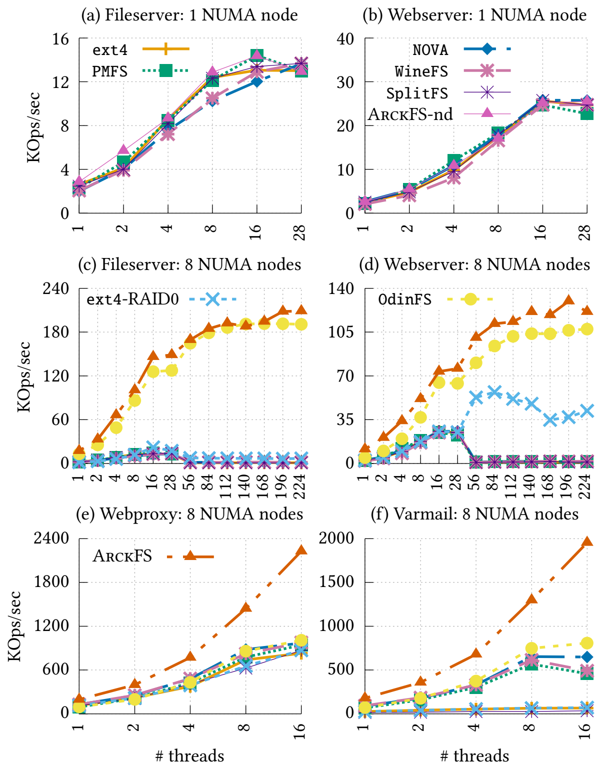

**Figure 9.** Filebench results. ArckFS consistently outperforms other evaluated file systems in various workloads. The one NUMA node results of Webproxy and Varmail are similar to those with eight NUMA nodes and thus omitted.

| Throughput (ops/ms) | ext4 | NOVA | WineFS | ArckFS | ArckFS-nd |
|---|---|---|---|---|---|
| Fill 100K | 1.23 | 2.53 | 2.60 | 3.81 | 2.71 |
| Fill seq | 135 | 210 | 239 | 419 | 561 |
| Fill sync | 17 | 189 | 211 | 291 | 378 |
| Fill random | 123 | 196 | 219 | 343 | 452 |
| Read random | 93 | 131 | 142 | 144 | 174 |
| Delete random | 148 | 217 | 245 | 494 | 603 |

**Table 5.** Performance of LevelDB with the evaluated file systems. `ext4(RAID0)` always underperforms `ext4`, and is thus omitted. We cannot evaluate other file systems since they do not implement (functional) `mmap()`.

Due to direct NVM access and efficient metadata operations, ArckFS significantly outperforms others, including `OdinFS`, with workloads performing many metadata operations and small file accesses. Specifically, for Webproxy and Varmail, ArckFS outperforms others by 2.2× to 8.0× and 2.4× to 34.2×, respectively.

**LevelDB.** We evaluate LevelDB by running `db_bench` with the default setup; `db_bench` runs with one thread, the value size is 100 bytes, and there are one million objects in the database. As shown in Table 5, ArckFS outperforms all the evaluated file systems across all the workloads. ArckFS outperforms the second-best performant file system: `WineFS`, by

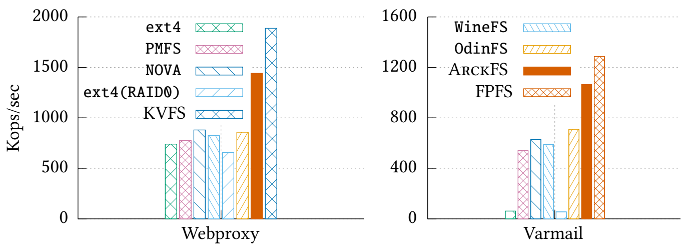

**Figure 10.** Customized file systems enabled by Trio: KVFS and FPFS further outperform ArckFS.

up to 3.1× and `ext4` by 1.5× to 17×. With `Fill100K`, ArckFS-nd is 29% slower than ArckFS. This is because `Fill100K` performs large file writes, and ArckFS benefits from the access parallelization enabled by the opportunistic delegation (§4.5). On the other hand, other workloads mostly perform small file accesses, where the opportunistic delegation incurs the striping and communication overhead (§6.2). As a result, ArckFS-nd outperforms ArckFS by 21% to 34%.

**Customization.** We design two customized file systems: KVFS and FPFS (§5) using the Trio architecture. Figure 10 demonstrates the performance benefits of customization. We extend Filebench with a key-value interface to support KVFS. We create a directory depth of 20 in Varmail to stress path resolution. Both workloads run with eight threads. In Webproxy, KVFS avoids managing file descriptors and file indexes, further outperforming ArckFS by 1.3× and others by 2.9×. In Varmail, FPFS supports full path indexing and thus further outperforms ArckFS by 1.2× and others by 21×.

**Summary.** ArckFS consistently outperforms other NVM file systems in macrobenchmarks that cover a wide range of file system use cases. The flexible customization enabled by Trio can further improve application performance.

## 7 Other related work

Trio follows Exokernel [25, 30] to provide applications with secure direct access to storage devices and a minimal abstraction for customization. Unlike Trio, Exokernel assumes a conventional disk and thus still requires kernel mediation for disk accesses. Furthermore, Exokernel cannot enable sharing among customized LibFSes and cannot enforce the metadata integrity of a POSIX-like file system. Arrakis [41] also assumes a conventional disk, leverages hardware extension to partition the disk statically, and assigns applications a portion of the storage device for direct access. Thus, accessing a file in another application's region requires inter-process communication. Trio resolves the four way tension among direct access, customization, sharing, and security on the emerging NVM storage device.

**State separation.** `ScaleFS` [18] decouples an in-memory file system from an on-disk file system using operation logs. Unlike `ScaleFS`, Trio only has a single layer of file systems: LibFSes and LibFSes directly accesses NVM, thereby avoiding the disadvantages of a log-based design (§2.3.1). `NOVA` [49] and Flatstore [19] maintain logs on NVM and maintain indexes in DRAM to improve performance. Trio generalizes this design philosophy, applies it to userspace NVM file systems, and solves different problems.

**Verifying file system consistency.** SQCK [28] is a file system checker based on SQL to achieve simplicity and better functionality than e2fsck. Trio's integrity verifier performs similarly to Recon [26], which pioneers enforcing metadata integrity online and locally.

## 8 Conclusion

This paper presents Trio, a new userspace NVM file system architecture that unleashes its performance potential and ensures metadata integrity. By separating the file system state into a single commonly shared core state and private auxiliary state in each component, Trio simultaneously allows LibFSes to directly access NVM, applications to flexibly customize its LibFSes without special privileges or affecting other applications, and untrusted applications can securely share a file with its private LibFSes. Leveraging Trio, we design ArckFS, a generic POSIX-like file system with a careful design to achieve low latency, high throughput, and excellent scalability for both data and metadata operations. Our extensive evaluation shows that ArckFS constantly outperforms existing NVM file systems by several times to orders of magnitude, while the customized file systems enabled by Trio further outperform ArckFS. Our artifact is publicly available at https://github.com/vmexit/trio-sosp23-ae.

## Acknowledgments

We sincerely thank our shepherd, Haryadi Gunawi, and the anonymous reviewers for their insightful feedback. We also thank Yuval Tamir and Changwoo Min for their helpful discussions. Zhang and Kannan were supported by funding from NSF grants CNS-1910593 and CNS-1730043 (experimental platform).

## References

[1] 3D XPoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[2] Compute Express Link 2.0 White Paper. https://b373eaf2-67af-4a29-b28c-3aae9e644f30.filesusr.com/ugd/0c1418$_1$4c5283e7f3e40f9b2955c7d0f60bebe.pdf.

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

[4] dm-stripe. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/striped.html.

[5] Exim Internet Mailer. https://www.exim.org/.

[6] ext4(5) — Linux manual page. https://man7.org/linux/man-pages/man5/ext4.5.html.

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

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

[9] fsck.ext4(8) - Linux man page. https://linux.die.net/man/8/fsck.ext4.

[10] HPE Persistent Memory. https://www.hpe.com/us/en/servers/persistent-memory.html.

[11] Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[12] Last week Intel killed Optane. Today, Kioxia and Everspin announced comparable tech: Rumors of storage-class memory's demise may have been premature. https://www.theregister.com/2022/08/02/kioxia$_e$verspin$_p$ersistent$_m$emory/.

[13] Memory Protection Keys. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html.

[14] Samsung Memory-Semantic SSD. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022.

[15] Small Files, Big Foils: Addressing the Associated Metadata and Application Challenges. https://blog.cloudera.com/small-files-big-foils/.

[16] The Challenge in Big Data is Small Files. https://blog.min.io/challenge-big-data-small-files/.

[17] util-linux. https://github.com/util-linux/util-linux.

[18] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai 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, October 2017.

[19] 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 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[20] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, February 2021.

[21] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 ACM SIGMOD/PODS Conference*, Xi'an, Shaanxi, China, May 2021.

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

[23] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo 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, October 2019.

[24] 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 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, April 2014.

[25] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole J. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[26] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, San JOSE, CA, February 2012.

[27] Bill Gervasi. A Persistent CXL Memory Module with DRAM Performance. In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/persistent-cxl-memory-module-dram-performance.

[28] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008.

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

[30] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.

[31] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay 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, October 2021.

[32] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay 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, October 2019.

[33] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, October 2021.

[34] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. Viyojit: Decoupling battery and dram capacities for battery-backed dram. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 2018.

[35] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[36] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, Rome, Italy, May 2023.

[37] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2015.

[38] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory. In

*Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2022.

[39] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[40] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022.

[41] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[42] Suchitra Raman and Steven McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proceedings of the 10th ACM SIGCOMM*, Cambridge, MA, August–September 1999.

[43] Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A Cross-layered Direct-Access File System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, November 2020.

[44] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 1992.

[45] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to Get More Value From Your File System Directory Cache. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[46] 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 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, April 2014.

[47] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Virtual, October 2020.

[48] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, April 2019.

[49] 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)*, Santa Clara, CA, February 2016.

[50] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudof. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[51] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven 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, February 2020.

[52] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[53] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The Full Path to Full-Path Indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, CA, February 2018.

[54] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. MadFS: Per-File virtualization for userspace persistent memory filesystems. In *Proceedings of the 21th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2023.

[55] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. Odinfs: Scaling PM performance with Opportunistic Delegation. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022.