Rethinking Tiered Storage: Talk to File Systems, Not Device Drivers

Jiyuan Zhang
University of Illinois
Urbana-Champaign, IL, USA
jiyuanz3@illinois.edu

Jongyul Kim
University of Illinois
Urbana-Champaign, IL, USA
jyk@illinois.edu

Chloe Alverti
University of Illinois
Urbana-Champaign, IL, USA
xalverti@illinois.edu

Peizhe Liu
University of Illinois
Urbana-Champaign, IL, USA
peizhel2@illinois.edu

Weiwei Jia
University of Rhode Island
Kingston, RI, USA
weiwei.jia@uri.edu

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

Abstract

Different storage technologies motivate the development of specialized file systems tailored to specific device types. A tiered file system aggregates such device types into a single file system. We argue that the current practice of developing tiered file systems tends to lag behind that of device-specific file systems because, inherently, developers are burdened with addressing multiple device types simultaneously, rather than specializing. We propose to solve this problem using Mux, a new tiered file system that accesses different device types indirectly through device-specific file systems, rather than directly through device drivers. Despite introducing an additional indirection layer, we show that Mux significantly outperforms Strata, a research tiered file system, because it utilizes specialized production-ready file systems. Compared with direct access to per-device file systems (with no tiering), Mux adds a worst-case read latency overhead of 6.6% to 87.3%, and a write throughout overhead of 1.6% to 3.5% across devices. We contend that Mux's separation of tiering and specialization concerns enables progressive evolution and flexible integration of heterogeneous storage devices.

CCS Concepts

• **Software and its engineering → File systems management**; • **Information systems → Hierarchical storage management**.

Keywords

Tiered Storage, File System, Operating System

This work is licensed under a Creative Commons Attribution 4.0 International License.

HOTOS '25, Banff, AB, Canada

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1475-7/2025/05
https://doi.org/10.1145/3713082.3730383

ACM Reference Format:
Jiyuan Zhang, Jongyul Kim, Chloe Alverti, Peizhe Liu, Weiwei Jia, and Tianyin Xu. 2025. Rethinking Tiered Storage: Talk to File Systems, Not Device Drivers. In *Workshop on Hot Topics in Operating Systems (HOTOS '25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3713082.3730383

1 Introduction

The landscape of modern storage is undertaking a disruptive change. The emergence of new storage technologies, such as persistent memory [13], CXL SSD [4, 5, 29], and others [8, 10, 35], are producing faster, larger, and cheaper storage devices. These devices offer new bandwidth, latency, and capacity features, such as near-memory speed, byte addressability, and cache coherence. Accordingly, many new, specialized file systems are being actively developed to fully utilize these new storage tiers [11–14, 20, 27, 30, 33, 34].

New devices are commonly integrated into *heterogeneous storage hierarchies* [17, 18, 21, 29] for cost efficiency and practical constraints (e.g., hardware acquisition and deployment). Tiered file systems [16, 19, 26, 31, 32] are a common design for heterogeneous storage. They expose the storage hierarchy as a single device to the host system and leverage the characteristics of each tier (device) internally to dynamically migrate files and blocks across tiers, minimizing latency and maximizing throughput while exposing large capacity.

We argue that the current practice of developing tiered file systems cannot keep up with the pace of storage innovation. First, their development inevitably lags behind device-specific file systems, as it involves integrating new device support and new tiering policies on every storage hierarchy extension. Second, it has become increasingly hard to cover all possible device configurations found in real-world storage infrastructures into one monolithic tiered file system.

In this paper, we propose to solve this problem using Mux, a new tiered file system that accesses different device types

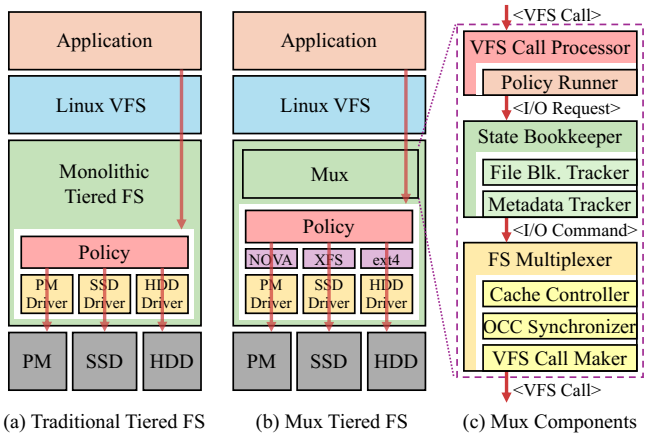


Figure 1: Traditional tiered file systems, Mux (our work) and its internal components.

indirectly through device-specific file systems, rather than directly through device drivers. As shown in Figure 1, Mux slots between the Virtual File System (VFS) and device-specific, native file systems; it realizes tiering policies (e.g., data placement, migration, and replication) and delegates I/O requests to native file systems. The design leverages the opportunity that native file systems are highly optimized and can be directly used as building blocks of a tiered file system. Different from existing meta file systems that combine native file systems such as OverlayFS [6], MergerFS [7], and UnionFS [24], Mux exposes the entire hierarchy as a single device to the host and transparently dispatches reads and writes to native file systems as per tiering policies. Mux distributes an individual file across tiers; it bookkeeps which file system stores the file blocks. The block distribution can happen both synchronously at allocation time (e.g., by append) and asynchronously by block-level data migration.

The design treats *extensibility*—seamless integration of new device types—as a first-class principle by decoupling tiering from device management. To integrate new devices, dedicated file systems can be plugged directly into the stack through a well-defined interface (e.g., Linux VFS), without modification. To extend tiering policies, Mux offers a modular interface (via kernel module or eBPF) for users to register tiering rules or device profiles. We contend that the separation of concerns—tiering and device specialization—would enable progressive evolution and flexible integration of heterogeneous file and storage systems.

Mux must address correctness challenges. Given the distribution of a file across file systems, which participating file systems are in charge of maintaining file metadata up-to-date? This metadata ownership challenge is unique, as a monolithic tiered file system stores a single instance of all attributes. We opt for distributing the metadata maintenance. We introduce *metadata affinity* to assign each metadata attribute to a file system owner. For example, the file system that stores the last byte of the file is the owner of the logical file size and the file system that performed the last update is the owner of the last modified timestamp. Mux bookkeeps which file system is the owner of an attribute and lazily synchronizes participating file systems. For metadata that cannot have a single owner such as disk consumption, Mux manages them across all related file systems.

Mux must also guarantee that data movement (e.g., for migration) across file systems does not interfere with user access—users' updates on blocks during migration are not lost or overwritten. The problem is also faced by monolithic tiered file systems and is commonly solved by locking the file or blocks in flight [16, 31, 32]. Differently, in Mux, migrations involve multiple native file systems, but no universal lock among them exists. Instead of global locks, we opt for optimistic concurrency control [15]. Mux maintains a *versioning* counter per file to detect potential conflicts based on version differences, therefore minimizing the impact of conflict checking on the critical path.

Despite introducing an additional indirection layer, Mux can directly benefit from device-specific optimizations of native file systems. This benefit can outweigh the indirection overhead in practice. We compare Mux which uses NOVA for persistent memory (PM), XFS for SSD, and Ext4 for HDD, with Strata [16], a research tiered file system. Mux achieves 1.46x higher throughput for device I/O and 2.59x faster data migration over Strata, because it utilizes specialized production-ready file systems (e.g., device-friendly journaling and caching scheme). Compared with direct access to per-device file systems (with no tiering), Mux adds a worst-case read latency overhead of 6.6% to 87.3%, as well as a write throughput overhead of 1.6% to 3.5%, across NOVA, XFS, and Ext4 on PM, SSD, and HDD, respectively.

We describe the design of Mux and its preliminary results, and discuss a few open problems. A full-fledged implementation of Mux is in progress.

2 Mux Design

We describe the Mux design that composes individual file systems into a tiered file system. The overarching goal is to aggregate the advantages of underlying device-specific file systems, while offering extensible tiering policies.

Mux must:

- manage the asynchronous distribution of data blocks (of files) across heterogeneous storage devices,
- guarantee the consistency of file metadata and data,
- dispatch user requests as individual per-device I/O operations and merge their results into unified responses,

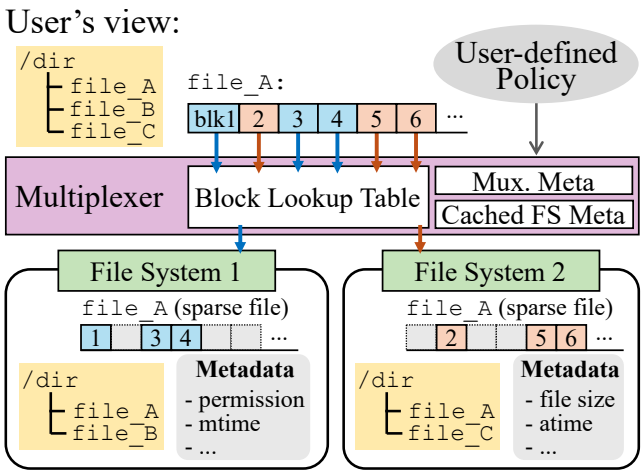


Figure 2: An overview of Mux.

- provide a user interface to specify or extend tiering rules,
- orchestrate shared file caching,
- eliminate races between user requests and asynchronous data movement (e.g., for migration).

Our performance goal is to match or exceed traditional "monolithic" tiered file systems, given the same tiering policy.

Mux slots between the VFS layer and device-specific file systems (Figure 1). Figure 2 shows the components of Mux and how file metadata and data are mapped to device-specific file systems.

The main challenge is the crossing of file system boundaries. In Mux, a file is typically distributed across multiple file systems, creating challenges for its indexing and metadata management. At the same time, data movement across file systems must not race or interfere with user requests, but the global locking is missing. Mux introduces a set of global metadata to bookkeep file distribution and devise efficient synchronization mechanisms.

2.1 Interface

Mux provides a uniform namespace to user applications. The underlying file systems selected by a user are mounted at different mount points. Mux merges these mount points in a single directory tree and presents it to user applications (Figure 2). A file can be distributed across multiple file systems; the same file name exists in different file systems. The design is inspired by OverlayFS [6] which overlays one file system on top of another, presenting a merged directory view to applications. In OverlayFS, all data updates are applied to the topmost file system. We extend this design to enable finer-grained and dynamic selection of the underlying file systems, including write operations.

To add a new device and the corresponding file system, the user only needs to mount the new file system and register it with Mux, along with a policy to manage it. To remove a device, data must be migrated first. Adding or removing a device can be done at runtime.

Mux is presented to the VFS layer as a standalone file system, making the OS send file operations to Mux through the existing VFS interface. Upon receiving a file operation, Mux splits it based on the block-to-underlying file system mappings, if required. Mux sends the split requests to device-specific file systems by calling the same VFS function that invokes it, but with different file handles, lengths, and offsets.

User-Defined Policy. Mux decouples tiering policies from file system implementation. It exposes an interface for users to specify policies on data placement and user request dispatching. All the placement and migration policies in existing tiered file systems [16, 19, 26, 31, 32] can be expressed using simple functions. For example, the data placement policy of TPFS [32] can be simply implemented by a function that returns different device IDs based on the I/O size, synchronicity, and access history. Currently, the policy is encoded as a kernel module or an eBPF extension so the policy functions can be directly called.

2.2 File and Data Block

Mux can support block-level data distribution and place data flexibly as per user-defined policies. For example, LRU-based or Hot-Cold block-level data migration schemes employed by existing tiered file systems [16, 31, 32] can be easily supported by Mux.

Block-level data distribution requires Mux to maintain the mapping from a block to the underlying file systems (a file system's internal index is invisible to Mux). We use a mapping table, named *Block Lookup Table*, to locate a block of a file, as shown in Figure 2. Mux maintains a per-file block lookup table and tracks in which device the recent version of a block is stored. Since the table maps file offsets to devices, that are small in size, we use an extent tree as a high-performance data structure.

Note that a file can be stored on multiple devices as a result of load balancing and data migration. Mux leverages sparse files to preserve the file offset of a block across different underlying file systems (Figure 2). It enables Mux to avoid an additional translation while preventing space waste from allocating the entire file.

2.3 Metadata

Managing file blocks across file systems creates a unique challenge. Which file system should keep the metadata attributes up to date, given that multiple ones may participate in metadata-update operations (e.g. appends)? We opt for

metadata multiplexing and introduce the notion of *metadata affinity*. For each metadata attribute, there is an *affinitive file system* at any given point in time, that holds the most up-to-date value for the attribute. This design reduces small writes due to metadata updates and avoids waiting on slow devices for synchronization.

For example, when the file is created, the host file system is the affinitive for all metadata. Later, if another file system performs an append guided by Mux (e.g., tiering the new blocks to multiple devices) the file system that allocated the last block of the operation becomes the affinitive for the file size attribute. During a write operation, the file system that overwrites the last block of the operation becomes the affinitive for the last update timestamp. During a read operation, the file system that fetches the last block of the operation becomes the affinitive for the last access timestamp, etc. Mux bookkeeps the affinitive file system per attribute, to orchestrate incoming requests.

Mux caches metadata attributes in a *collective inode*, avoiding fetching metadata from different file systems (which is expensive in the multiplexed design).

Mux maintains its own metadata like block lookup table, file affinity table, etc. The space overhead of such metadata is marginal. For example, in a block lookup table, one byte per 4 KB of user data is sufficient with a simple byte array, leading to less than 0.025% of space overhead.

2.4 Data Movement

In heterogeneous storage, data movement is critical to performance and cost efficiency. Traditional tiered file systems realize transparent migration by implementing intra-file system locking. It enables the migration and application's file operations to be performed atomically with respect to each other. However, such locking is insufficient in Mux. In Mux, blocks are moved from one file system to another, which requires a synchronization mechanism across the file systems. Moreover, applications' file operations are an external operation to underlying device-specific file systems. Hence, Mux requires a new synchronization mechanism to support transparent migration.

We design the synchronization mechanism using optimistic concurrency control [15], which is called OCC Synchronizer. Our insight is that data movement does not change the content of the data; so, a data movement process is considered successful if the content of the data remains unchanged throughout the process. With this principle, we introduce a version number to each file to identify the potential conflicts. The version number is incremented at the start and end of a data movement process, allowing Mux to detect the possibility of a conflict while handling an application's file write by comparing the version number before and after the write operation and checking a migration flag on the file. If a potential conflict is detected, Mux searches for blocks that have changed during migration. If any blocks are found to have been modified, Mux retries the migration of those blocks. If all tasks that ran during the migration process are conflict-free, the migrated block will be atomically committed and made visible in the block lookup table. If OCC Synchronizer keeps failing to migrate the block, Mux will resort to a lock-based migration. This scheme minimizes the critical path of user requests and enables the parallel execution of migration without pessimistic blocking or lock contention.

Since a data block migrated using OCC Synchronizer is only atomically made visible when proven to be conflict-free, it avoids data corruption due to data race. Also, since any tasks that could generate a conflict are checked before committing the block, and those tasks are proven not to have modified the migrated data block, the migrated data is also guaranteed to be up-to-date. If any conflict occurs, the migrated data block will be dropped and overwritten in place in the next migration attempt. This ensures that there are no side effects from failed attempts. OCC Synchronizer also guarantees that the migration process will be completed in a finite amount of time, due to the limited number of retries, thus ensuring the replication lag is bounded.

2.5 Caching

Mux includes a cache manager to use Storage Class Memory (SCM) devices (e.g., persistent memory) as caching devices. Recent studies [9, 16, 19] show that using SCM for caching improves storage performance, as SCM has a large capacity and is accessible via memory operations. Existing tiered file systems [16, 31] assume SCM-based caches.

In the Mux design, while each file system may use DRAM as its page cache, the cache cannot be shared across devices. Moreover, as storage continues to grow, DRAM is difficult to scale. Using SCM devices to offload DRAM page caches helps alleviate the scalability problem. SCM file systems [12–14, 20, 27, 33, 34] commonly use the DAX (Direct Access) interface, where memory mapping a file provides direct access to the physical storage, minimizing software overhead. Mux uses DAX to implement data cache to minimize software overhead.

Mux can create one file for all caches, which helps reduce the overhead of managing multiple files as well as disk fragmentation. Alternatively, Mux can preallocate the cache file to ensure cache availability and reduce block allocation overhead. Mux uses DAX memory mapping for the cache file. If the caching memory pool is depleted, the data can be written back or evicted. We use Multi-generational LRU [3] for cache replacement, which is also the algorithm Linux uses for its page caches.

3 Preliminary Results

We report the performance of an early prototype of Mux, in comparison to Strata, a traditional tiered file system. We also report the overhead of Mux over native device-specific file systems when accessing files directly (without tiering).

3.1 Performance

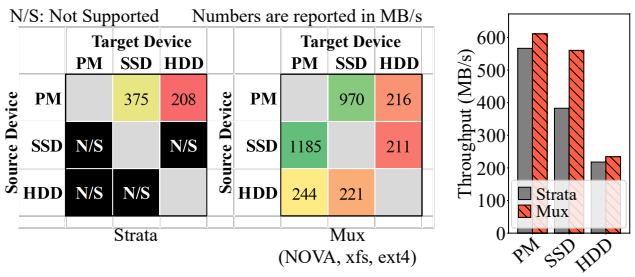
We use a storage hierarchy that consists of Intel Optane Persistent Memory 200 (as PM), Intel Optane SSD DC P4800X (as SSD), and Seagate Exos X18 (as HDD), paired with Intel Xeon 6346 CPU and 256GB DRAM. We use a prototype of Mux that utilizes specialized production-ready file systems, including NOVA [27] for PM, XFS [25] for SSD, and Ext4 [22] for HDD. We compare Mux with Strata [16], a research tiered file system. Mux and Strata use the same tiering policy to manage the storage hierarchy.

Strata follows the traditional tiered file system design. It is designed as a monolithic file system that manages all three devices internally. Our Mux prototype implements the tiering mechanism internally, but delegates the I/O to NOVA, XFS, and Ext4. We implement a simple LRU policy that evicts cold data to the slower device if no space left on faster devices, and promotes data back upon access.

We focus on data migration, which is arguably the most basic tiering feature. Mux directly supports data migration between any pair of devices (Figure 3a). As VFS abstracts out device-specific details, supporting a migration path takes a single line of code to invoke the migration function. However, this is not the case for Strata, where adding a path requires manually matching the threading model, block size, and call context of the paired devices. If two devices are not wired, data migration between them either is unsupported (N/S) or must be relayed via other devices. Among all six migration pairs, Strata supports *only two* (Figure 3a). Also, it supports neither SSD to HDD demotion nor any kind of promotion.

We measure the maximum throughput of data migration between each device pair by continuously writing data to saturate upper-tier devices, forcing data migrations. As shown in Figure 3a, the throughputs of inter-device data migration in Mux are much higher than those in Strata. Specifically, the throughput of PM to SSD migration is increased by 2.59x. We also measure the I/O throughput of individual devices. We run Strata's microbenchmark [2] with 90GB random writes and measure the throughput when the I/O request is always directed to the target devices. As shown in Figure 3b, the I/O throughput of Mux is 1.08x, 1.46x, and 1.07x higher for PM, SSD, and HDD respectively over Strata.

We attribute the performance advantage of Mux over Strata for two reasons. First, Mux leverages the device-specific optimizations provided by the underlying file systems. For example, we observe that Strata first writes data to a log



(a) Extensibility **(b) Performance**

Figure 3: Extensibility and performance of Strata and Mux. Strata uses static routing among tiers and only supports PM to SSD and PM to HDD data movement; Strata has lower performance compared to device-specific file systems on a single PM/SSD/HDD device.

on persistent memory and then digests the log to actual file blocks on final storage devices. However, such logging is not necessary on persistent memory devices. The optimized file system we use for PM, namely NOVA, is different in that it writes data directly to the file blocks using DAX and uses the CLFLUSH instruction to guarantee data consistency. Hence, Strata's design results in write amplification and performance degradation. Second, unlike Strata, Mux uses independent file systems to drive devices. In Strata, the file extent tree that contains both block offset and device index has to be partially locked during block-level data migration, which may prevent access to blocks that do not require migration. In contrast, since Mux works on top of device-specific file systems and has its own separate metafile storage, locking file blocks for migration will not affect access to other blocks. We acknowledge that Strata is a research system and its performance may not represent an *ideal* tiered file system. On the other hand, Mux uses production-ready, highly optimized device-specific file systems, which shows the advantage of the Mux design—the extensibility allows Mux to directly benefit from device-specific optimizations of well-developed file systems.

3.2 Overhead

Mux explores a tradeoff: introducing an additional file system layer above the device-specific file systems to get the performance benefit of per-device specialization for free. One may suspect an 100% overhead of Mux, as it doubles the work of file systems. To understand the worst case of the indirection overhead, we run a microbenchmark that repeatedly *reads* one single byte from a 10GB file randomly. Compared to the native file systems—NOVA, XFS, and Ext4 (with no tiering), Mux increases the latency by 52.4%, 87.3%, and 6.6% on PM, SSD, and HDD, respectively.

We also run a microbenchmark that repeatedly writes four megabytes to a file sequentially to evaluate write throughput overhead. Compared to the native file systems, Mux decreases the throughput by 1.6%, 2.2%, and 3.5% on PM, SSD, and HDD, respectively.

Note that some of the above overhead may also appear in an ideal tiered file system; how to quantify the Mux overhead over an ideal implementation remains our future work.

4 Discussion

With the initial prototype that shows the promises, we are developing a full-fledged Mux. We discuss several challenging problems we encounter and ideas to address them.

Crash Consistency. Currently, the crash consistency properties of Mux are composed of those of the participating file systems. Mux sends fsync requests to all the file systems that are responsible for a given file and synchronizes the completion of the fsync operations. Upon a crash, Mux relies on each participating file system to recover the data blocks it stores. However, different file systems have different crash consistency guarantees [23] and the overall crash consistency of a file is affected by defects in any participating file system, assuming no data replication. We believe that a much stronger crash consistency guarantee can be designed for Mux by utilizing the diversity of device-specific file systems and by the opportunity for data replication across devices.

Feature Imparity of File Systems. A common thread in the Mux design is the difference between different file systems, despite that they all implement the VFS interface. For example, file systems often have special features that the others do not. Some file systems extend metadata attributes in uncommon ways. Oftentimes, even for the same metadata attribute, its semantics can vary (e.g., FAT records timestamps with a two-second granularity [1]). We currently do not consider any extended metadata attributes or special features. How to enable users to effectively manage these system-specific features and metadata is an open problem.

Improving The I/O Scheduler. High-performance I/O often requires I/O schedulers to maximize resource utilization and minimize contention. The I/O scheduler should identify request types, estimate their costs, and reorder them to optimize performance [28]. We currently use a simple scheduling algorithm based on device profiles (performance characteristics and feature sets). We expect more intelligent schedulers to be developed, that consider dynamic states such as load spikes on devices as well as instantaneous I/O activities for garbage collection and in-device DRAM flushing to SSDs.

Configuring Mux. As the Mux design can easily integrate many existing file systems, an emerging problem is how to find the best configuration of file systems for a given workload or a given set of storage devices. Sharing Mux among multiple applications may also require scheduling schemes that support priority, deadline, and/or quota, which may dispatch I/Os and accessed data blocks to file systems with different performances, or ensure that high-priority tasks are not impeded by reordering and splitting requests.

Distributed Mux. One ambitious idea is to extend Mux in a distributed manner. By designing a Mux-to-Mux interconnection (e.g., through Remote Procedure Call) at the Mux layer and a distributed tiering policy, it is possible that a set of machines mounting traditional file systems can be integrated into a distributed storage system. Certainly, to realize that, we will need to address many open problems, some of them are: (1) how to achieve transparency across the OS boundary? (2) how to support dynamic joining and leaving of nodes? and (3) how to handle latency variations and network faults? We plan to start with attaching networked file systems as one of the underlying file systems.

Acknowledgement

We thank our shepherd Dan Tsafrir for the extensive technical discussion that reshaped the paper. We thank Swaminathan Sundararaman and Nam Sung Kim for their valuable technical discussion. The paper is supported in part by NSF CNS-1956007, CNS-2145295, SHF-2348066, an IIDAI grant, and the Wing Kai Cheng Fellowship.

References

[1] Microsoft FAT Specification. https://academy.cba.mit.edu/classes/networking_communications/SD/FAT.pdf, Aug. 2005.

[2] Microbenchmarks - ut-osa/assise. https://github.com/ut-osa/assise/tree/master/bench/micro, Apr. 2021.

[3] Multi-generational LRU: the next generation. https://lwn.net/Articles/856931/, May 2021.

[4] Samsung Plans to Start Large-Scale Production of CXL Devices Soon. https://semiconductor.samsung.com/us/news-events/news/samsung-demonstrates-new-cxl-capabilities-and-introduces-new-memory-module-for-scalable-composable-disaggregated-infrastructure-at-memcon-2024/, Mar. 2024.

[5] Intel Has Incorporated CXL into Its Modern Processors. https://www.intel.com/content/www/us/en/support/articles/000059219/processors.html, Jan. 2025.

[6] Overview of the Overlay File System. https://docs.kernel.org/filesystems/overlayfs.html, Jan. 2025.

[7] trapexit/mergerfs. https://github.com/trapexit/mergerfs, Apr. 2025.

[8] Anderson, P., Aranas, E. B., Assaf, Y., Behrendt, R., Black, R., Caballero, M., Cameron, P., Canakci, B., De Carvalho, T., Chatzieleftheriou, A., Storan Clarke, R., Clegg, J., Cletheroe, D., Cooper, B., Deegan, T., Donnelly, A., Drevinskas, R., Gaunt, A., Gkantsidis, C., Gomez Diaz, A., Haller, I., Hong, F., Ilieva, T., Joshi, S., Joyce, R., Kunkel, M., Lara, D., Legtchenko, S., Liu, F. L., Magalhaes, B., Marzoev, A., Mcnett, M., Mohan, J., Myrah, M., Nguyen, T., Nowozin, S., Ogus, A., Overweg, H., Rowstron, A., Sah, M., Sakakura, M., Scholtz, P., Schreiner, N., Sella, O., Smith, A., Stefanovici, I., Sweeney, D., Thomsen, B., Verkes, G., Wainman, P., Westcott, J., Weston, L., Whittaker, C., Wilke Berenguer, P., Williams, H., Winkler, T., and Winzeck, S. Project Silica: Towards

Sustainable Cloud Archival Storage in Glass. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)* (Oct. 2023).

[9] Anderson, T. E., Canini, M., Kim, J., Kostić, D., Kwon, Y., Peter, S., Reda, W., Schuh, H. N., and Witchel, E. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)* (Nov. 2020).

[10] Bornholt, J., Lopez, R., Carmean, D. M., Ceze, L., Seelig, G., and Strauss, K. A DNA-Based Archival Storage System. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)* (Mar. 2016).

[11] Chen, Y., Lu, Y., Zhu, B., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Shu, J. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)* (Feb. 2021).

[12] Chen, Y., Shu, J., Ou, J., and Lu, Y. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. *ACM Transactions on Storage (TOS) 14*, 1 (Apr. 2018).

[13] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)* (Apr. 2014).

[14] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A., and Chidambaram, V. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)* (Oct. 2019).

[15] Kung, H.-T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (1981), 213–226.

[16] Kwon, Y., Fingler, H., Hunt, T., Peter, S., Witchel, E., and Anderson, T. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)* (Oct. 2017).

[17] Lee, T., Monga, S. K., Min, C., and Eom, Y. I. Memtis: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)* (Oct. 2023).

[18] Li, H., Berger, D. S., Hsu, L., Ernst, D., Zardoshti, P., Novakovic, S., Shah, M., Rajadnya, S., Lee, S., Agarwal, I., Hill, M. D., Fontoura, M., and Bianchini, R. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)* (Mar. 2023).

[19] Lin, Z., Xiang, L., Rao, J., and Lu, H. P2CACHE: Exploring Tiered Memory for In-Kernel File Systems Caching. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC '23)* (July 2023).

[20] Lu, Y., Shu, J., Chen, Y., and Li, T. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)* (July 2017).

[21] Maruf, H. A., Wang, H., Dhanotia, A., Weiner, J., Agarwal, N., Bhattacharya, P., Petersen, C., Chowdhury, M., Kanaujia, S., and Chauhan, P. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)* (Mar. 2023).

[22] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Ottawa Linux Symposium (OLS 2007)* (June 2007).

[23] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[24] Quigley, D., Sipek, J., Wright, C., and Zadok, E. Unionfs: User- and Community-Oriented Development of a Unification File System. In *Proceedings of the 2006 Ottawa Linux Symposium (OLS '06)* (July 2006).

[25] Sweeney, A. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference (USENIX '96)* (Jan. 1996).

[26] Wu, K., Guo, Z., Hu, G., Tu, K., Alagappan, R., Sen, R., Park, K., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)* (Feb. 2021).

[27] Xu, J., and Swanson, S. 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)* (Feb. 2016).

[28] Yang, S., Harter, T., Agrawal, N., Kowsalya, S. S., Krishnamurthy, A., Al-Kiswany, S., Kaushik, R. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Split-level I/O scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)* (Oct. 2015).

[29] Yang, S.-P., Kim, M., Nam, S., Park, J., Choi, J.-Y., Nam, E. H., Lee, E., Lee, S., and Kim, B. S. Overcoming the Memory Wall with CXL-Enabled SSDs. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC '23)* (July 2023).

[30] Zhan, Y., Hu, H., Yang, X., Wang, S., Cao, Q., Jiang, H., and Yao, J. RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths. In *Proceedings of the 15th ACM Symposium on Cloud Computing (SoCC '24)* (Nov. 2024).

[31] Zheng, S., Hoseinzadeh, M., and Swanson, S. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)* (Feb. 2019).

[32] Zheng, S., Hoseinzadeh, M., Swanson, S., and Huang, L. TPFS: A High-Performance Tiered File System for Persistent Memories and Disks. *ACM Transactions on Storage 19*, 2 (2023), 1–28.

[33] Zhou, D., Aschenbrenner, V., Lyu, T., Zhang, J., Kannan, S., and Kashyap, S. Enabling High-Performance and Secure Userspace NVM File Systems with the Trio Architecture. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)* (Oct. 2023).

[34] Zhou, D., Qian, Y., Gupta, V., Yang, Z., Min, C., and Kashyap, S. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)* (July 2022).

[35] Zhou, J., Dong, M., Wang, F., Zen, J., Zhao, L., Fan, C., and Chen, H. Liquid-State Drive: A Case for DNA Block Device for Enormous Data. In *23rd USENIX Conference on File and Storage Technologies (FAST '25)* (Feb. 2025).