Ethane: An Asymmetric File System for Disaggregated Persistent Memory

Miao Cai, *College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics;* Junru Shen, *College of Computer Science and Software Engineering, Hohai University;* Baoliu Ye, *State Key Laboratory for Novel Software Technology, Nanjing University*

https://www.usenix.org/conference/atc24/presentation/cai

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

Ethane: An Asymmetric File System for Disaggregated Persistent Memory

Miao Cai[†], Junru Shen[‡], Baoliu Ye[§]
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics [†]
College of Computer Science and Software Engineering, Hohai University [‡]
State Key Laboratory for Novel Software Technology, Nanjing University[§]

Abstract

The ultra-fast persistent memories (PMs) promise a practical solution towards high-performance distributed file systems. This paper examines and reveals a cascade of three performance and cost issues in the current PM provision scheme, namely expensive cross-node interaction, weak single-node capability, and costly scale-out performance, which not only underutilizes fast PM devices but also magnifies its limited storage capacity and high price deficiencies. To remedy this, we introduce Ethane, a file system built on disaggregated persistent memory (DPM). Through resource separation using fast connectivity technologies, DPM achieves efficient and cost-effective PM sharing while retaining low-latency memory access. To unleash such hardware potentials, Ethane incorporates an asymmetric file system architecture inspired by the imbalanced resource provision feature of DPM. It splits a file system into a control-plane FS and a data-plane FS and designs these two planes to make the best use of the respective hardware resources. Evaluation results demonstrate that Ethane reaps the DPM hardware benefits, performs up to 68× better than modern distributed file systems, and improves data-intensive application throughputs by up to 17×.

1 Introduction

Distributed file systems (DFSs) are the backbone of modern data center storage. To meet the unprecedented performance demands posed by data center applications [17, 20, 77], distributed file systems heavily rely on high-speed storage devices like persistent memories [12, 36, 41, 49, 72]. In contrast to the large, cheap, slow storage devices (e.g., solid-state or hard-disk drives), persistent memory is a disruptive storage technology with three distinctive features, namely ultra-fast speed (~300 ns latency), limited storage capacity (≤512 GB per DIMM slot), and expensive price ($3.27/GB). In the current monolithic data centers [61], every server machine is equipped with a number of PM modules, dubbed *symmetric* PM architecture in this paper. This egalitarian PM provision, however, leads to a cascade of performance and cost issues.

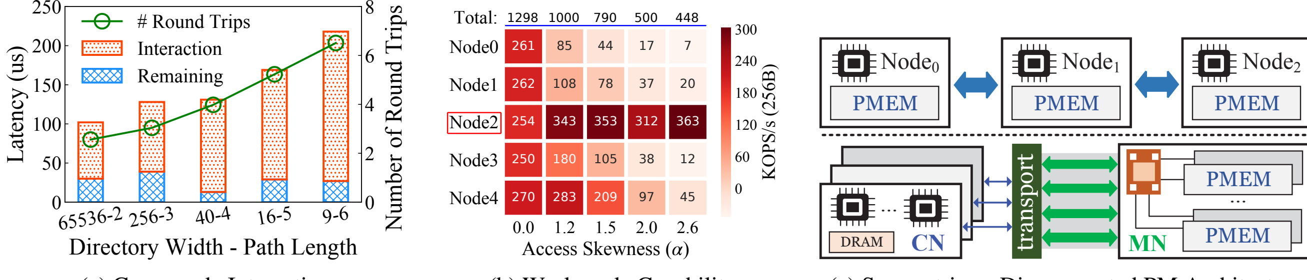
For symmetric PM architecture, file system data are scattered over a cluster of machines. When serving a client request, the server node has to interact with other nodes, resulting in excessive, expensive network round-trips. More severely, when distributed data meets non-uniform access patterns, the load imbalance problem arises and the server node storing the hot files inevitably becomes a performance bottleneck, crippling the overall system performance. To remedy the bottlenecked node, administrators have to purchase more machines to amortize the hotspot pressure. Unfortunately, besides the necessary PM devices, additional expenses have to be paid for encapsulated processors and other peripheral devices, which significantly increases the total cost of ownership (TCO) for data center vendors.

To summarize, this hardware usage magnifies PM's drawbacks of limited capacity and high price as well as underutilizes precious PM resources. Furthermore, our analysis in Section §2.1 reveals that expensive cross-node interaction, weak single-node capability, and costly scale-out performance caused by the symmetric PM architecture result in unpredictable request latency, degraded overall performance, and high monetary costs for distributed file systems, making them hardly meet the stringent Server Level Objectives (SLOs) in the regime of "Killer Microsecond" [16, 23].

To tackle these issues, we propose Ethane [1], a file system built on disaggregated persistent memory. The persistent memory disaggregation is a key enabling technique for the next-generation high-performance data center [34, 43, 61, 65, 75]. DPM separates CPU and PM resources and assembles them into dedicated compute nodes (CNs) and memory nodes (MNs) connected with fast data connectivity technologies (e.g., RDMA [31, 40, 60] and CXL [44, 45, 52]), which delivers both surpassing large storage capacity and high aggregated bandwidth in a cost-efficient manner. The DPM architecture is appealing due to the fast evolution of surging high-speed memory and network technologies [31, 44, 45, 52, 60, 73].

To drive the DPM system, we depolymerize the compound

[1] Ethane (C_2H_6) is an organic chemical compound whose structural formula resembles a DPM system with RDMA-connected CNs and MNs.



(a) Cross-node Interaction (b) Weak-node Capability (c) Symmetric vs Disaggregated PM Architecture

Figure 1: Performance Issues in Symmetric PM File Systems and Disaggregated PM Architecture. *(a) demonstrates that the cross-node interaction occupies a large portion of the total execution time in CephFS [69]; (b) showcases the overall system performance of Octopus [49] is crippled due to single node performance limitation; (c) presents the symmetric and disaggregated PM architecture.*

file system architecture with a key insight. Particularly, the DPM features imbalanced resource provision between CNs and MNs. CNs yields superior computing capability than MNs whereas only owns a few gigabytes of DRAM. In contrast, MNs is equipped with tera- or peta-bytes of PMs but is supplied with less powerful processing units. This characteristic inspires us to design an asymmetric file system architecture, which splits file system functionalities into a control-plane FS and a data-plane FS, making the best use of respective strong computing and memory resources.

- The control-plane FS is responsible for handling complicated system control and management logic like concurrency control and crash consistency. Leveraging the centralized view of the shared MN, we delegate intricate control-plane FS functionalities to simplified, lightweight shared log [13–15, 25, 38, 48, 67]. For instance, linearizable system call execution is turned into a log ordering problem. Extracting file system semantics, we propose a variety of techniques to improve log insert scalability, reduce log playback latency, and achieve strong operation durability.
- The data-plane FS is responsible for storage management and processing data requests. It aims to harvest the large capacity and aggregated bandwidth benefits of parallel-connected MNs. Towards this end, we design a unified storage paradigm for translating a variety of dependence-coupled file system data structures into unified, access-disentangled key-value tuples and propose mechanisms to achieve parallel metadata and data paths.

We build Ethane for an RDMA-capable disaggregated PM system and evaluate it on an emulation platform with a rack of four Intel Optane DC persistent memory machines connected with a 100 GbE Mellanox switch. We compare Ethane with three modern distributed PM file systems: Octopus [49], Assise [12], and CephFS [69]. Evaluations show promising results. Ethane delivers much better NIC and PM bandwidth utilizations. It achieves up to 68× higher throughputs and up to 1.71× lower monetary costs with synthetic benchmarks. When running a replicated key-value store Redis Cluster [9] and a MapReduce application Metis [24], Ethane improves their performance by up to 16×.

To sum up, this paper makes the following contributions.

- We examine current PM use in distributed file systems and reveal three issues. To tackle these issues, we advocate disaggregating PM and propose an asymmetric file system architecture with a novel functionality separation.
- Leveraging the centralized view of the shared memory node, we define the control-plane FS based on shared log abstraction for efficient functionality delegation.
- To harvest the aggregated bandwidth, we design the data-plane FS as a key-value store with a unified storage paradigm and dependence-disentangled data access.
- We demonstrate the performance benefits and cost efficiency of prototyped Ethane with extensive experiments.

2 Background and Motivation

2.1 Symmetric PM Architecture

The commercialized PM device is a paramount storage technology to hunt the "Killer Microseconds" [16, 23] for its hundreds of nanoseconds of latency and large bandwidth. Hence, commodity distributed file systems [12, 33, 36, 41, 49, 72] extensively and intensively use PMs to fulfill the strict performance requirements for data center applications. From the hardware perspective, data centers manage resources in the unit of monolithic servers. Every server machine is full-featured which hosts both CPU and PM resources. From the file system perspective, this PM usage induces a series of correlated issues, as described below.

Expensive cross-node interaction. A distributed file system usually stores a large volume of application data [30, 56, 69]. In a symmetric PM architecture, data are scattered and managed by independent server nodes. These nodes are self-managed individuals that run customized, deep storage and network stacks and communicate using general-purpose RPCs [36, 63, 69, 72]. When a server node receives a client request, it has to interact with other nodes to serve the request. An interaction includes cross-node communications and a (meta-)data fetch from the target node. Considering current DFSs, general communication mechanisms and long data paths lead to a high end-to-end interaction latency.

We use file path resolution in CephFS [69] to describe and quantify interaction costs. CephFS partitions the namespace tree among a number of metadata servers (MDSs) [70]. When resolving a file path, the client accesses multiple MDSs to fetch directory entries (dentries) and inodes. We conduct an experiment that runs an RDMA-enabled CephFS with PM-based OSD storage. We use MDTest [7] to generate a large directory tree with 65535 entries across four MDSs. A client issues `stat` to access files in the namespace. Figure 1a shows the latency breakdown and the number of network round trips.

CephFS incorporates a message-based RPC and a BlueStore storage backend [11] with a layer of RocksDB, BlueFS, and PMDK. For a remote dentry read, the sender encapsulates the request in a message, copies and serializes data in the message buffer, and transmits it over the network transport (i.e., RDMA-over-RoCE). The receiver deserializes the message, loads data with the BlueStore and copies them to the NIC buffer, and transmits the response. As a result, a complete interaction includes costly data movements, encapsulation, de-/serialization and passes through several storage layers, which takes ~162 μs and occupies 60.24% of the component resolution time.

Furthermore, serialized component resolution design in CephFS incurs excessive sequential cross-node interactions. The linear growth of expensive interactions significantly affects the overall syscall latency. The interaction time even occupies 91.71% of the total syscall time when resolving a six-component path.

Weak single-node capability. Due to manufacturing restrictions, a machine only can be equipped with a few PM DIMMs [73] which limits the total PM capacity by up to a few terabytes. Moreover, commodity PM devices have a limited bandwidth. Performance studies show that only four parallel writers with an IO size of 256B saturate the bandwidth [32,73]. This small performance and storage upper bound raise serious concerns for production data-intensive applications due to their non-uniform access popularity [17, 20, 74]. Skewed data access causes a server node to easily become a bottleneck, crippling the overall file system performance.

Figure 1b demonstrates the load imbalance issue of a PM-based distributed file system, Octopus [49]. This experiment creates a set of 4KB files and distributes them over four server nodes. Ten clients issue read requests to four server nodes. The IO size is 256B and the data access popularity follows a Zipf distribution [20]. For a uniform request distribution (i.e., $\alpha = 0$), all server nodes deliver almost the same throughput. When α increases, Node2's throughput increases but this node is bottlenecked. More severely, other nodes' PM devices become underutilized and their throughputs drop dramatically. The total throughput also decreases significantly by up to two times. The weak node deficiency is an inevitable consequence of scattered data distribution, and this problem can hardly be resolved as hot data keeps frequently shifting and changing [20, 74].

Costly scale-out performance. To remedy the single-node weakness and keep pace with the exponential growth of application requirements, data center vendors have to purchase more PM machines. Unfortunately, the symmetric PM provision makes this scale-out paradigm costly and inefficient, especially for distributed data processing applications with high elastic resource requirements [24]. For example, Hadoop [1], a well-known MapReduce implementation, is designed with two procedures: *map* and *reduce*. Each procedure consists of two independent phases: a computing-intensive phase for data processing and an IO-intensive phase for data loading/writing.

Hadoop uses HDFS [63] as its primary storage system. Suppose an HDFS node is running the IO phase and its PM devices are under-provisioned. After adding a PM machine, other HDFS nodes are unable to enjoy the added capacity and performance benefits directly. Besides that, coupled CPUs in the new machine may be over-provisioned for the computing phase. Our evaluation §5.4 shows that this low elastic resource scaling significantly increases the total purchase budget and maintenance costs for MapReduce applications.

2.2 Disaggregated PM Architecture

To tackle aforementioned issues, decoupling PMs from monolithic machines and aggregating them in a dedicated memory pool is a promising solution. The disaggregated PM architecture enables efficient and cost-effective PM sharing with fast data access [43, 46, 51, 65, 78]. Depicted in Figure 1c, either a CN or an MN is a specialized machine that assembles blades of computing or memory resources, exhibiting much stronger hardware capability than a monolithic machine in the symmetric PM architecture. Moreover, the shared PM pool embraces the fast evolved data connectivity technologies like RMDA [60] and CXL [64]. It supports low-latency data connections with highly aggregated bandwidth. Finally, the DPM allows independent scaling of two types of hardware resources. The administrator could provision or de-provision a specific type of hardware resource flexibly and on-demand. Recently, the DPM architecture is propelled rapidly thanks to ultra-fast memory and high-speed network technologies [31, 44, 45, 52, 60, 73].

Besides performance and cost advantages, the DPM has a unique feature: *resource asymmetry*. In particular, CNs and MNs exhibit respective strong computation and memory capabilities. A CN is equipped with powerful processing units and limited memory capacity. Its small-sized memory only can be used for hot data caching or running performance-critical tasks locally. In contrast, MNs manage a large memory pool consisting of tens or even hundreds of PM modules [78]. However, every MN is only equipped with weak computing units (e.g., ARM SoC and ASIC) that support running necessary system tasks like memory scrubbing [43]. Realizing this characteristic, the challenge is how to design a file system to fully drive such distinctive hardware architecture.

3 Asymmetric File System Architecture

To respond to the question, we introduce an asymmetric file system architecture. Inspired by the resource asymmetry characteristic, we split the functionalities of a file system into two planes: (1) a control-plane FS which is responsible for managing and controlling file system states; (2) a data-plane FS which is responsible for storage management and processing data requests, and run these two FS planes on the CNs and MNs, respectively.

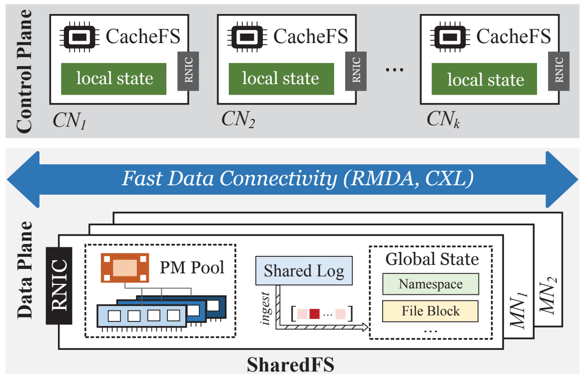


Figure 2: Asymmetric File System Architecture. *The control plane consists of a set of cacheFS instances running on computing nodes, whereas the data plane provides a centralized sharedFS atop the memory nodes.*

Separation of control and data plane: File systems abstract, define, and store various objects, such as dentries, inodes, and file blocks. These objects are classified into two categories: meta objects and data objects. File system operations manipulate these objects, e.g., a `chmod` changes the inode permission fields. Conventional DFS architecture is built upon the principle of separate object management [30, 63, 69], i.e., meta and data objects are stored and manipulated by different nodes. This design is well-suited for file systems built on symmetric PM architecture, whereas it is ill-suited for the DPM as neither a CN nor an MN has sufficient PM or CPU resources for object storage or manipulation.

We propose a design principle of *separating FS object manipulation from storage*. Guided by this principle, we split a file system into a control-plane FS and a data-plane FS. The data-plane FS stores both meta and data objects as well as provides efficient mechanisms to access them. The control-plane FS fetches objects from the data-plane FS and handles complex and intricate object manipulation logics, such as namespace query, crash consistency, and concurrency control.

Best use of hardware resource. The primary goal of this functionality separation is to make the best use of available hardware in each server node. The CNs exhibit superior computing capability. We deploy the control-plane FS on the CNs to handle complex and compute-intensive system management tasks. We incarnate the control-plane FS as a set of *cacheFSes* that run atop available CNs. Each cacheFS instance maintains a cached, partial view in its local, small DRAM.

On the other side, MNs provide a shared memory pool with PB-scale PM modules. Memory blades are parallel connected through high-performance NICs [34, 46] or CXL controllers [45, 52]. Because the MN offers a global view of the whole file system, we design the data-plane FS as a *sharedFS* which shards data over disjoint PM modules and parallelizes data access paths with hardware-provided parallelism.

Shared-log-based control-plane FS. The control-plane FS is built upon the shared log abstraction [13–15, 25, 38, 48, 67]. We identify the shared log is a good fit for the control-plane FS for two reasons. First, the memory node provides a centralized view for all compute nodes, which natively supports efficient data sharing. Second, implementing control-plane FS functionalities is complicated and requires considerably sophisticated techniques [12, 57, 69, 71]. The shared log provides an elegant and efficient means to achieve them simultaneously. We delegate the control-plane FS functionalities to the shared log and propose a range of techniques to support strong persistent guarantee, efficient concurrency control, and low-cost state coherence for cacheFS instances.

Access-disentangled data-plane FS. An endemic in conventional DFSs is entangled data paths, i.e., data access is tightly coupled with data processing inside file system operations [19]. For example, a path resolution resolves a number of path components. A component resolution includes a dentry read and many other coupled dentry processing like sanity checks. This sequential, entangled data path squanders the large DPM bandwidth and magnifies the latency inefficiency of the RDMA network. To deal with it, we propose a DPM-friendly data path which disentangles the data access from other coupled operations. It overlaps data access to reap the aggregated bandwidth of parallel-connected PM devices.

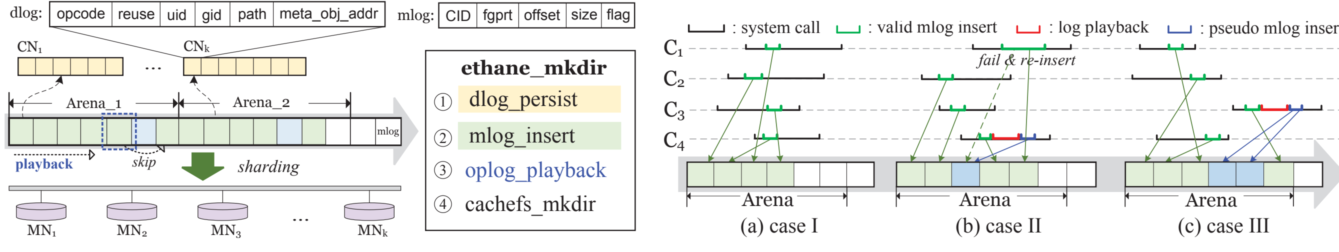
4 Ethane: Design and Implementation

Applying the architecture, we build Ethane, a file system for RDMA-enabled disaggregated persistent memory. We present the design of its control plane §4.1 and data plane §4.2, as well as the implementation §4.3.

4.1 Control-plane FS

The control-plane FS consists of a set of cacheFS instances. Every cacheFS maintains volatile, partial states. The local state is small, which is sufficient to reside in the small-sized local DRAM, and is volatile, which can be rebuilt from the remote sharedFS. The cacheFS mainly consists of two components: (1) a namespace cache which stores recently accessed namespace entries and is structured as a chain-based hash table; (2) a block cache which caches the data block metadata (e.g., remote address) and is organized as an AVL tree.

The core functionalities of the control-plane FS, such as cacheFS operation durability, concurrency control of cacheFS



(a) The Arena Layout and Syscall Execution Flow

(b) Three Cases of Operation Log Insertion

Figure 3: Log Arena Design

operations, and coherence among cacheFS instances, are delegated to the shared log. Illustrated with the example in Figure 3a, the following section elaborates on our delegation mechanisms and techniques for optimizing shared log persistence, insertion, and playback.

4.1.1 Delegating Durability to Log Persistence

We decouple the log persistence from the log ordering [25]. A syscall has an operation log (oplog) which includes a data log (dlog) and a meta log (mlog). Every cacheFS has a private PM region in the MN for storing dlogs. A dlog contains an opcode, a file path, credentials, the meta object address, and a reuse field used in the collaborative log playback. Moreover, there is a global log order array for storing mlogs. An 8-byte mlog packs a 12-bit cacheFS ID (CID), a 2-byte path fingerprint, a 26-bit dlog region offset, a 9-bit dlog size, and a 1-bit flag for indicating this oplog is associated with a `rename`/`symlink` syscall or other syscalls.

As shown in Figure 3a, the cacheFS creates a dlog and a mlog for a `mkdir` syscall. It first writes the dlog in the private region via a `RDMA_WRITE` (①). Then it uses another `RDMA_READ` to the queue pair issued the `RDMA_WRITE` in order to flush the MN's PCIe buffer [68]. Persisting the dlog ensures the data durability of this syscall. Leveraging the in-order delivery property provided by commodity RNICs [66], we issue these two RDMA requests simultaneously.

4.1.2 Delegating Linearizability to Log Ordering

The control-plane FS provides a compatible linearizability model instead of a relaxed consistency model for clients [48]. The shared log approach turns concurrent syscall execution into a sequential history of oplogs. The cacheFS writes the mlog in the global log order array (②). The mlog order in the order array reflects the order of corresponding syscall executions. Every cacheFS instance replays the same log sequence as if these syscalls take place locally. Thus, producing a sequence of mlogs with respect to linearizable syscall executions is the key to linearizable cacheFS design.

To achieve a valid log sequence, a naive solution is using `RDMA_CAS` to append mlogs to a list one by one [14, 15]. Imposing a strict order with `RDMA_CAS` is expensive. It is because modern RNICs use an internal lock to serialize concurrent `RDMA_CAS`es [40, 66]. High RNIC contention renders the mlog list tail to become a severe scalability bottleneck. Our insight is that producing a sequence of mlogs for linearizable syscalls does not require linearizable mlog append. We propose log arena mechanism which dramatically reduces mlog insert contention while still enforcing a valid log order with respect to linearizable syscall executions.

Figure 3a shows the mlog region is partitioned into a series of arenas. An arena consists of a number of slots for storing mlogs. An arena insertion consists of two steps: (1) a mlog insertion and (2) filling preceding empty slots. In step (1), every cacheFS randomly picks up an empty slot in the current active arena and uses an `RDMA_CAS` to insert the mlog and persist it with an `RDMA_READ` (case I in Figure 3b). In an ideal case, no preceding empty slots exist when C_1-C_4 finish arena insertions. Thus, step (2) is omitted and these arena insertions have no contention. Also, they generate a valid log sequence. The trick is that C_1-C_4 are concurrent cacheFS instances. Hence, there are no order restrictions for associated mlog insertions.

When a cacheFS finishes the arena insertion, it should ensure that there are no empty slots preceding the inserted $mlog_x$. Otherwise, if a subsequent cacheFS inserts a $mlog_y$ into one of those empty slots, $mlog_y$ precedes $mlog_x$ in the log history. The linearizability is violated. To prevent this, in case II, C_4 scans preceding slots and fills empty slots with pseudo mlogs. The pseudo mlog represents a null operation. The empty slot belongs to the in-flight C_1. C_1 and C_4 are concurrent instances. If C_4 precedes C_1, C_4's pseudo mlog insertion may cause C_1's insertion to fail. C_1 would re-insert the mlog.

Scanning and filling empty slots increases the arena insertion latency and causes contention for concurrent insert operations. We introduce two optimizations. First, the number of slots in an arena is set to be smaller than the number of concurrent threads. Thus, all empty slots are likely to be filled by threads and the pseudo log insertion rarely happens. Second, we perform step (2) after log playback (③), which creates a time window for those in-flight arena insertions. Both optimizations try to minimize the likelihood of empty slots for concurrent arena insertions.

In case III, C_3 is non-concurrent with the other three cacheFS instances. There are no empty slots after the other three cacheFSs insert mlogs. Thus, C_3's mlog locates behind their mlogs in the arena. When C_3 finishes mlog insertion, there exists two empty slots ahead of its mlog. It fills these

empty slots to complete the log history.

4.1.3 Delegating Coherence to Log Playback

Every cacheFS maintains a coherent state via replication. The coherence among replicated cacheFS instances is achieved via log playback. File system interfaces are not nilext [29]. Thus, to return a correct value, the syscall should externalize its effect and modify file system states immediately. Therefore, the cacheFS first forwards its local state to a newest one by scanning and replaying oplogs (③). Afterwards, it executes the `mkdir` locally and returns the execution result (④).

The non-nilext interface property forces the log playback to appear during the syscall execution path. To resolve the bottleneck, we propose two techniques: file-lineage-based log dependence check and collaborative log playback.

Log dependence check. Replaying a long sequence of logs significantly increases the total latency. To reduce the log playback sequence length, the cacheFS aims to play dependent logs. To this end, we need to answer two questions: (a) which oplogs are dependent? and (b) how to identify dependent oplogs quickly.

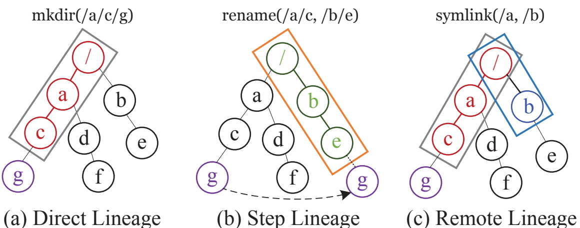


Figure 4: Direct, Step, and Remote Lineage

To respond (a), we validate the dependence of two file system operations based on *file lineage*. The direct lineage of a file f is defined as a set of files whose paths is a prefix of f's path. Figure 4(a) illustrates an example. `/a/c/g`'s direct lineage includes `/`, `/a`, and `/a/c`. The f's operation depends on operations whose files and directories belong to f's lineage. For instance, removing `/a` or disabling `/a`'s read permission causes `/a/c/g` to become inaccessible.

A directory rename operation changes f's existing lineage. Figure 4(b) shows that `rename(/a/c, /b/e)` moves files from `/a/c` to `/b/e`. Files in `/a/c` change their lineage. The new lineage is called *step* lineage. Dependence checking of oplogs behind the `rename`'s oplog uses the step lineage.

The symbolic link (symlink) adds a new file lineage. If a source directory is a symlink and points to a directory which does not belong to its lineage, the lineage of the target directory is the new lineage for children in the source directory. We call the new lineage *remote* lineage. Figure 4(c) shows that `/a/c` is a symlink and points to `/b`. `/a/c/g` has a remote lineage `/b`. Dependence checking of oplogs behind the *symlink*'s oplog uses both the direct and remote lineage.

To answer (b), we design a mlog skip table to reduce the log playback range. Each cacheFS has a volatile, private mlog skip table. Every file has a corresponding table entry which records the final slot position in the global log order array during last playback. It helps skip logs which have been replayed in the last playback. As shown in Figure 5, the cacheFS calculates the lineage for the target file `/a/b/c` (①), queries the skip table with the fingerprint of every file path (②), and finds associated playback ranges of every file. The final playback range is the union of all ranges.

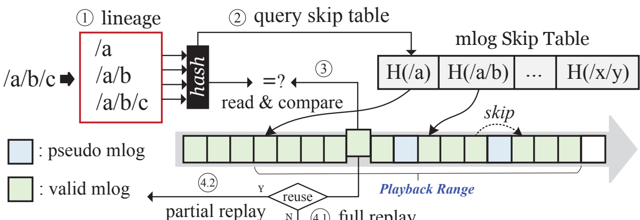


Figure 5: Fast, Collaborative Log Playback

During log playback, the cacheFS reads mlogs one by one and checks their dependence (③). In particular, we calculate the path fingerprints in the lineage and compare each of them with the path fingerprint stored in the mlog. If one equals, this oplog is dependent. Thus, the cacheFS reads the dlog and performs the associated operation to update cacheFS states. In addition, if an mlog is associated with a `rename` or a `symlink` syscall, the associated step or direct and remote lineage is used for checking log dependence.

Collaborative log playback. The log playback performs history operations to derive a coherent cacheFS state. A complete file syscall has a long execution path. The collaborative log playback accelerates syscall execution by reusing partial execution results of other log playback routines. Specifically, almost all metadata syscalls are composed of two parts: a file path walk and the final file modification (e.g., changing file credentials). The file path walk is lengthy as it consists of a series of path component resolutions which occupies a large portion of the total execution time [19, 50].

We aim to reuse the path walk result of other log playbacks. This reusing mechanism is feasible. Suppose the current log playback contains a log. This log has a deterministic order in history as well as a set of dependent logs. If another cacheFS has played this log before, these dependent logs already have been replayed in its local state. The file path walk of this oplog in these two playbacks produces the same result.

We add a *reuse* field in the dlog. A set field indicates that this log has been played before. Hence, the current playback routine performs a partial log replay ((4.2)) by fetching the path resolution result and modifying the file directly. Otherwise, it performs a complete log replay ((4.1)).

4.2 Data-plane FS

The data-plane FS provides a shared, DPM-friendly storage layer. It unifies data management for diverse file system structures with a key-value-based storage paradigm and a vector

access interface. Furthermore, it provides disentangled, parallelized (meta-)data access paths to harvest DPM bandwidth.

4.2.1 Data Storage Paradigm

Metadata and data in file systems are managed with various data structures. For example, dentries are organized in a namespace tree [57]. Ethane designs a unified key-value storage paradigm for (meta-)data indexing and management. The key-value storage paradigm is advantageous for PM file systems. It is expressive and provides efficient support for structured file system data [37,58]. Also, it offers fine-grained and easy-to-use interfaces which effectively exploits PM byte-addressability to avoid access amplification [42].

Type	Key	Value	
Meta Object	/a/b	[obj_1_addr, obj_0_addr]	obj_1
	/a/b/c	[obj_2_addr, obj_1_addr]	obj_2
	/a/hardlink	[obj_2_addr, obj_0_addr]	
	/a/b/symlink	[obj_3_addr, obj_1_addr]	obj_3
Data Section	[obj_2_addr, start_addr, section_size]	extent_addr	section / extent

Figure 6: Key-value Data Storage Paradigm

Translating FS objects to KV tuples. The data-plane FS mainly includes three types of objects: a superblock, meta objects, and data sections. The superblock records global file system states. Every file or directory has a meta object which stores its metadata, including file type, file size, and full path, etc. Figure 6 shows that every meta object is associated with a key-value tuple. The key is the unique full path. The value is a combination of the meta object address itself and the meta object address of its parent directory. A hard link to a target file has no meta object. Its value stores a pointer to the target file's meta object. In contrast, a symlink has an independent meta object.

The data-plane FS uses extents to organize data blocks. An extent is a mapping from contiguous logical blocks to contiguous physical blocks. To translate a file offset to a physical block number —*file mapping*, file systems often use extent tree [47, 53]. This translation method is unfriendly for DPM due to a cascade of pointer chasings during tree traversal. To overcome this, we propose a data section-based file mapping design.

Every file has a logical contiguous linear space. A data section represents an aligned range of linear space and has three fixed sizes: 1 GB, 2 MB, and 4 KB. Moreover, a data section is associated with an extent and its range is inclusive to that extent range. A large extent may have several data sections with different sizes. For example, an extent with a mapping range of [0, 2113536] has a 2 MB section and four 4 KB sections. A data section has a key of a concatenation of three fields: the memory address of its file's meta object, the section start address, and the section size. The value is a backward pointer to the associated extent. For a file mapping, we find the data section first and use the pointer to get the associated extent. The detailed file mapping procedure is presented later §4.2.2.

Hash-based data management. We use cuckoo hash tables to manage key-value tuples for each type of file system objects. We choose cuckoo hash table because its constant number of slot probes per lookup facilitates designing parallel data search. We use global instead of per-file block management [53], i.e., key-value tuples of all files' data sections are managed by global hash tables. The key space of a hash table is split. A sharding of key-value tuples is stored in a cuckoo hash table. The cuckoo hash table is organized as a linear array of slots. We stripe the linear array across all available MNs.

Access interface. The data-plane FS provides a vector-based key-value *get* interface: `int vec_kv_get(key_t *k_vec, val_t *v_vec)`. This interface is *vectorized* which accepts a bunch of keys. It is beneficial for batched file system operations. For example, a file path walk needs to search a collection of meta objects for component resolutions. These meta object lookups can be performed at once via this interface.

In addition, this interface is *approximate* which returns a number of possibly correct values. Due to hash collisions, a hash table lookup needs to validate keys. To saturate the DPM bandwidth, this interface delays key validation and instead returns all possibly correct values for the caller at once. The caller at the CN side filters out desired values afterward. In addition, the data-plane FS also provides a vector-based key-value *put* interface.

4.2.2 Data Path Disentanglement

The data path in traditional file system calls includes entangled, sequential data access and data processing. We introduce disentangled data path and demonstrate it with two examples. It separates the data access from the data processing, so as to leverage the vector lookup interface to saturate the aggregated DPM bandwidth and hide RDMA network latency.

Parallel, pipelined hash lookup. The vector lookup interface is implemented via parallel, pipelined cuckoo hash lookups. The cuckoo hash introduces two slot probes per lookup and there are no memory access dependence for these two slot probes. Exploiting this feature, we propose a pipelined lookup mechanism. A slot probe includes a sequence of computational tasks, which calculates hash value and target MN, etc, and a one-sided `RDMA_READ`. We overlap these computational tasks with the remote memory access for pipelined slot probe.

Furthermore, our cuckoo hash table supports optimistic concurrency control [28]. Every slot contains a version number. Readers are lock-free. It reads and compares the versions of two slots before and after reading the data. Only if these two versions are unchanged, this lookup succeeds. Otherwise the reader retries. Therefore, a lookup requires two rounds of slot probes at least. Only the first round of slot probes is

optimized with the pipeline mechanism.

File path walk. A metadata syscall first performs a path walk and then modifies the meta object. The file path walk is an iterative process which consists of a number of path component resolutions. Traditionally, each path component resolution includes a dentry fetch (*data access*) and a series of coupled operations (*data processing*) like permission checks.

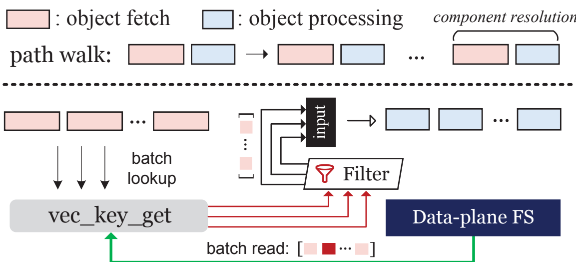


Figure 7: Disentangled File Path Walk

We decompose the path walk into a batch of dentry lookups and remaining operations. A dentry lookup equals a meta object search in our data-plane FS. Suppose a syscall `unlink(/a/b/c)` and the associated path `/a/b/c` contains three components. Its file path has three prefix paths. In Figure 7, to find associated meta objects for three prefix paths, the cacheFS issues three lookups via one `vec_kv_get` invocation. The `vec_kv_get` computes six hash values for three keys, performs six parallel, dependence-free hash lookups, and returns six lookup results. The return values may contain false meta objects, i.e., their file paths do not belong to `/a/b/c`'s lineage. To filter out correct objects quickly, we perform a swift sanity check on returned meta objects before exact filename comparisons.

Assume we check returned meta objects for the prefix path `/a`. We compare the parent directories of returned meta objects with the root directory and abandon these objects with an incorrect parent. We use the correct meta object as a new parent directory and repeat this sanity check with the next prefix path `/a/b`. Note that the root directory has an empty parent directory. After finding out all correct meta objects, it performs remaining operations at once.

File data read. For a `read(int* fd,void* buf,off_t offset,size_t count)`, the file system performs a file mapping by traversing the extent tree, visiting tree node entries (*data access*), performing a binary search (*data processing*), and comparing the extent range with the requested range (*data processing*). Then it reads block data (*data access*) and repeats this process until all requested data are fetched. This data read consists of a number of sequential extent tree lookups and block reads, resulting in a relatively long read path.

Our data-plane FS decomposes the data IO path into a series of disentangled file mappings and parallel data reads, as shown in Algorithm 1. First, our file mapping performs batched data section lookups to find extents. The endpoint `addr` is initialized as the left point of the lookup range `[offset, offset+count]`. Because the target data section size is unknown, we calculate the start addresses of three possible data sections in line 4-6. Next, we compute three key tuples in line 7 and send data section lookup requests via a `vec_kv_get` invocation in line 8.

Algorithm 1: read(int* fd,void* buf,size_t count,off_t offset)

```
1  uint64 union_min = offset, union_max = 0, addr = offset, i = 0;
2  struct* extents[N]; struct block* blks[M]; vec_t v_vec;
3  while union_min >= offset && union_max <= (offset+count) do
4      uint64 section1_start = ALIGN_DOWN(addr/SIZE_1GB);
5      uint64 section2_start = ALIGN_DOWN(addr/SIZE_2MB);
6      uint64 section3_start = ALIGN_DOWN(addr/SIZE_4KB);
7      key_t* k_vec = vectorize_key(section1_start, section2_start,
         section3_start); // get a vector of keys
8      vec_kv_get(k_vec, v_vec);             // batch section lookup
9      extents[i] = filter_sections(v_vec);
10     union_max += extents[i]->range_size; // extend the union range
11     addr += extents[i]->range_size; i++; // update lookup endpoint
12 blks = get_blocks(extents); // get blocks in all extents
13 buf = read_blocks(blks, offset, count); // parallel block reads
```

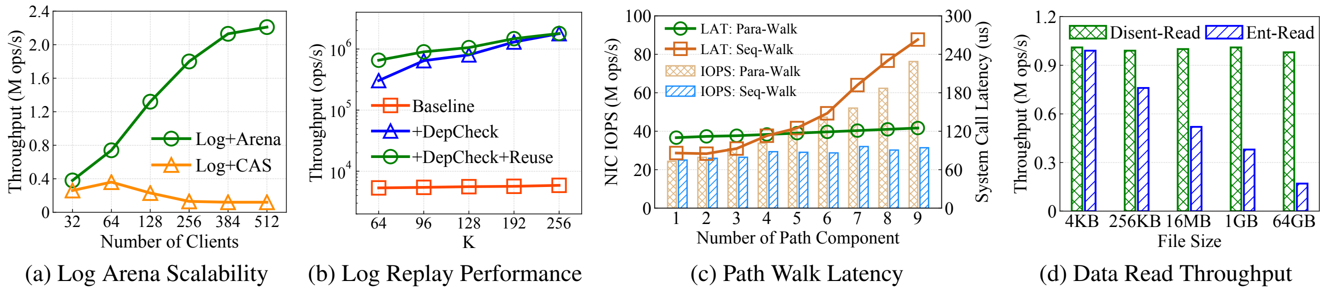
The `vec_kv_get` performs DPM-friendly, batched lookups and returns a vector of data sections. We filter out the correct section by validating candidate sections' ranges and get the associated extent in line 9. After that, we extend the union range of found extents in line 10 and update the lookup endpoint in line 11. If the union range cannot include the requested range, we perform another extent lookup. Otherwise, we get all blocks for found extents in line 12. The data-plane FS stripes a file in the unit of extents across PM devices in the memory pool, which facilitates parallel data R/W onto file blocks belonging to disjoint extents. Hence we parallelize data block reads in line 13.

4.2.3 Log Ingestion

The sharedFS ingests shared logs to update its states. The log ingestion is split into two phases. First, every CN runs a log ingestion worker and suppose there are N workers in total. Logs belonging to the same file are dependent. Every worker i scans all shared logs and gathers those logs whose $fingerprint\%N == i$ in its working set. It ensures that dependent logs are processed by the same worker. The worker fetches and replays operation logs to forward cacheFS states. When finishes replaying logs, it translates data in the namespace cache and block cache into corresponding key-value tuples and feeds these key-value data to sharedFS via a vector *put* invocation. Second, the sharedFS ingests these data by creating, inserting, or updating associated key-value tuples.

4.3 Implementation

We develop Ethane prototype from scratch. Its source code is available at https://github.com/miaogecm/Ethane.git. It consists of 10910 lines of C code. The CN runs the Linux operating system to provide POSIX-compatible interfaces, efficient resource management, and data protection. The cacheFS is implemented as a user-level library which includes 4922 lines of C code.



(a) Log Arena Scalability (b) Log Replay Performance (c) Path Walk Latency (d) Data Read Throughput

Figure 8: Control-Plane FS and Data-Plane FS Evaluation Results

Every cacheFS instance runs as a state machine which replays logs to transit local states. Multiple cacheFS instances are isolated from each other. CacheFS instances are classified into two categories: external cacheFSs and internal cacheFSs. An external cacheFS is linked with a client and serves user requests. Internal cacheFSs are used for log ingestion. There are no clients for internal cacheFS instances.

The MN has less computation power and is unable to support executing a full-fledged operating system kernel. Instead, every MN runs a thin sharedFS daemon which is responsible for PM pool management and log garbage collection. Ethane runs on the reliable connected RDMA transport. We use the ZooKeeper [2] for managing namespace and maintaining configuration information for CNs and MNs.

We assume the sharedFS belongs to the trusted computing base. Every cacheFS instance runs in the user space. Its states and data are volatile and can be rebuilt based on the sharedFS and shared logs. Any data corruption or stray writes [26, 57, 76] that happened in the cacheFS or client applications are unable to affect the data integrity of sharedFS. The log ingestion of internal cacheFS instances is performed by dedicated kernel threads at the CN.

5 Evaluation

This section tries to answer three questions: (1) Is Ethane friendly to disaggregated persistent memory architecture? (§5.1-5.2) (2) Does Ethane perform better than conventional distributed file systems? (§5.3) (3) How does Ethane perform with real-world data-intensive applications? (§5.4)

We set up two platforms with a rack of four blade servers for emulating disaggregated PM and symmetric PM. Every blade server has two NUMA nodes and is equipped with two Intel Xeon Gold 5220 CPU @ 2.20 GHZ, 128 GB (4×32 GB) SK Hynix DDR4 DRAM, 512 GB (4×128 GB) Intel Optane DC Persistent Memory (DCPMM), a 512 GB Samsung PM981 NVMe SSD, and two Mellanox ConnectX-6 100 GbE NICs. Every blade server runs Ubuntu 18.04 and is connected to a 100 Gb Ethernet Mellanox switch.

Table 1 lists the hardware configurations and estimated prices of disaggregated PM and symmetric PM systems. Hardware device prices are collected from Amazon and HPE CDW websites. In the experiment, a disaggregated PM system consists of two CNs and two MNs and a symmetric PM system includes four symmetric nodes (SNs). The total prices of two systems are $14764 and $15156, respectively.

Table 1: Hardware Configuration of Disaggregated and Symmetric PM Systems

	CPU	DRAM	PMEM	SSD	NIC	Price
CN	32 cores	8 GB DDR4	-	512 GB NVMe SSD	2×ConnectX-6 NIC	$3919
MN	1 core	8 GB DDR4	4×128 GB DCPMM	512 GB NVMe SSD	2×ConnectX-6 NIC	$3463
SN	16 cores	2×32 GB DDR4	2×128 GB DCPMM	512 GB NVMe SSD	ConnectX-6 NIC	$3789

We compare Ethane with CephFS [69], Octopus [49], and Assise [12]. These three DFSs run on the symmetric PM system. We configure an RDMA-enabled CephFS with BlueStore as OSD storage backend. The OSD uses PMDK to manage DCPMM. The CephFS runs four MDS daemons and four OSD daemons. We pin a pair of MDS and OSD daemons to an SN. Due to the limited number of physical cores, experiments also use a coroutine library [6].

5.1 Control-plane FS Evaluation

The control-plane FS functionalities is delegated to the shared log. We evaluate its performance by analyzing three shared log techniques: log persistence, log arena and log playback.

Log persistence latency. We measure the oplog persistence latency with different log size. An oplog consists of a mlog and a dlog. We evaluate three dlog sizes (64B, 1024B, 4096B) by varying file path length. Inserting and persisting a mlog/dlog need one `RDMA_CAS`/`RDMA_WRITE` and one `RDMA_READ`. An `RDMA_CAS`/`RDMA_READ` and `RDMA_WRITE` latency is 4.8 μs and 3.2 μs. These two RDMA requests are transmitted in parallel. Thanks to this optimization, the 8-byte mlog insertion and persistence latency takes 5.48 μs. Inserting and persisting a small- and medium-sized dlog costs 4.32 μs and 5.53 μs. The 4096B dlog insertion and persistence takes 7.46 μs.

Log arena scalability. This experiment studies log insert scalability. Every client has a private working directory and repeats creating files in this directory. We compare our log arena approach (Log+Arena) with a `RDMA_CAS`-based solution (Log+CAS). The number of slots in an arena is set to the half of the number of clients. The experiment varies the number of clients. Figure 8a demonstrates that Log+Arena scales much

better than Log+CAS. Its throughput increases linearly when the number of clients increases.

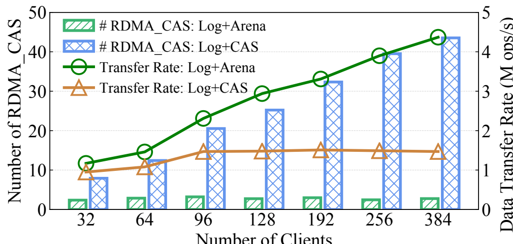


Figure 9: # `RDMA_CAS` and PCIe Data Transfer Rate

We collect the number of `RDMA_CAS` operations and report them in Figure 9. The Log+CAS introduces much more atomic RDMA operations than Log+Arena. In contrast, the Log+Arena incurs a few `RDMA_CAS` operations. Moreover, Log+CAS has a low PM bandwidth utilization. `RDMA_CAS` causes heavy NIC lock contention which prevents throughput increment. We use Intel PMWatch tool [5] to measure the number of write operations received from the memory control, i.e., `ddrt_write_ops`. The RNIC accesses the DCPMM via PCIe bus and the on-chip PCIe controller forwards requests to the memory control. Thus, the increase rate of `ddrt_write_ops` approximates the PCIe data transfer rate. The Log+Arena achieves a much higher PCIe data transfer rate. Its peak rate is 3× higher than Log+CAS, leading to a better PM bandwidth utilization.

Log playback efficiency. Our log playback consists of two techniques: lineage-based log dependence check and collaborative log playback. We first evaluate the effectiveness of the log dependence check. In the experiment, every client creates files in the directory `/ethane-<X>` where $1 \le X \le K$. Experiments vary K from 64 to 256. Before a client creates a file, it must replay all dependent logs. Hence, a small K causes more dependent logs.

The baseline replays logs one by one without any dependence check or reusing optimizations. Figure 8b shows that the baseline throughput is only 5 Kops/s. The log dependence check optimization is effective. When K increases, the number of dependent logs reduces. It filters out a large number of unrelated logs during log replay. The +DepCheck improves the throughput by up to 1.7 Mops/s. The log reusing mechanism brings more performance benefits. Because all `creat` syscalls share a common prefix path, the collaborative log playback utilizes this to accelerate the execution path. It performs 42.21% better than +DepCheck on average.

5.2 Data-plane FS Evaluation

The data-plane FS incorporates DPM-friendly, disentangled data paths. This section analyzes two representative data paths: file path walk and file data read.

Path walk latency. This experiment preloads a Linux-4.15 source code repository. Then it creates 256 clients and each client accesses a file in the directory tree via `stat`. We change the file path length and measure the syscall latency. We compare the parallel path walk design in Ethane with the traditional sequential path walk. Figure 8c demonstrates that Para-Walk delivers a much more stable latency than that of Seq-Walk. The Seq-Walk is non-scalable. Its total path walk latency is proportional to the number of path components. Its path walk latency at 9-component takes 263 μs which is 3.09 × higher than that of 1-component.

The Para-Walk decouples the component access from component processing. It utilizes the vector lookup interface to perform parallel component lookups, which hides the RDMA latency and delivers high network bandwidth utilization. To confirm it, we profile the NIC IOPS at the receiver side. Figure 8c shows that Para-Walk achieves up to 2.43× higher IOPS than that of Seq-Walk.

Data read throughput. This experiment creates a client to perform random data read to a file. The IO size is set to 4 KB. We change the file size from 4 KB to 64 GB. We compare two data path designs: disentangled read path (Disent-Read) and entangled read path (Ent-Read). The entangled read path uses extent tree-based block management. Figure 8d shows that the Disent-Read performs much better than the Ent-Read, especially for a large file size. When the file size is 64 GB, its throughput is 5.76× higher than that of the Ent-Read. This is because the file mapping dominates the large file read time.

Table 2: # Pointer Chasing and % File Mapping Time

File Size	# Pointer Chasing		% File Mapping Time	
	Disent-Read	Ent-Read	Disent-Read	Ent-Read
4 KB	2.0	2.0	28.2%	49.7%
256 KB	2.0	2.76	31.4%	62.2%
16 MB	2.0	3.89	29.9%	77.8%
1 GB	2.0	4.38	29.7%	80.2%
64 GB	2.0	5.41	30.5%	86.7%

When the file size is large, the extent tree is high. As a result, the Ent-Read needs to traverse many tree levels to find a data block. To analyze performance overheads deeply, Table 2 reports the number of pointer chasings during two data paths. It shows that the Ent-Read requires 5.41 pointer chasings in extent tree traversal for a 64 GB file read. The total file mapping time occupies 86.7% of the total time. Thanks to the data section-based block management, our Disent-Read only incurs two pointer chasings per file read regardless of file size changing. Moreover, it only spends 28.2%-30.5% of the total time in file mapping.

5.3 Macrobenchmark Performance

End-to-end latency. We measure the end-to-end latency of system calls. Eight clients run on four nodes and send `mkdir` and `stat` requests to backend file system servers. Figure 10 reports measured latencies. In the first experiment, every client creates directories in `/ethane-<X>` where X=*rand(1,K)*. We vary K from 1 to 10. As shown in Figure 10a, deep software stack and distributed namespace tree in CephFS causes over

eight hundred microseconds of `mkdir` latency. Octopus delivers approximately 60 μs latency. Assise achieves 31.27% lower latency than that of Octopus on average owing to its client-local NVM design. Ethane delivers the similar latency as Assise when K is small. Fortunately, when K increases, the number of dependent logs decreases. The log playback latency in Ethane effectively reduces. It outperforms Assise by up to 33.54%.

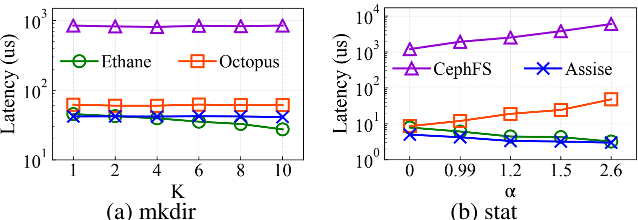


Figure 10: System call Latency

In the second experiment, every client accesses files via `stat`. The file access pattern follows a Zipfian distribution with a parameter α. A large α indicates a skewed access pattern. Both CephFS and Octopus suffer from the load imbalance issue. Their `stat` latencies increase linearly when α becomes bigger. Assise replicates hot data in client-local NVMs. Thus, skewed file access improves its performance. Similarly, the namespace cache design in cacheFS also helps avoid the load imbalance issue for Ethane.

Metadata scalability. We use MDTest [7] to evaluate metadata performance. We run an MPICH framework [8] to generate MDTest processes across server nodes. Before the experiment, every process creates a private directory hierarchy for each client. Then, every client creates two million files, accesses them, and removes these files. Figure 11 shows the metadata performance of four file systems.

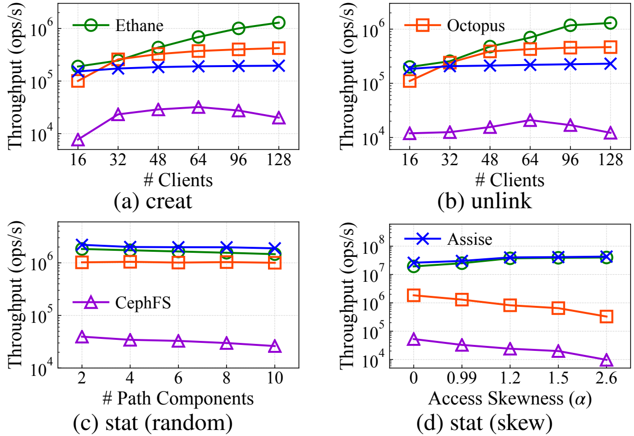


Figure 11: MDTest Performance

For file creation and removal, Ethane outperforms all other three distributed file systems. Octopus only assigns one worker per data server. It is unable to process massive client requests efficiently, which causes severe weak node capability issues. The total throughput of Octopus stops increasing at 48 clients. CephFS yields orders of magnitudes of lower throughputs than Ethane. The MDS in CephFS relies on OSDs to store metadata. A `creat` or `unlink` syscall causes frequent cross-node communication between MDS and OSD. Moreover, the BlueStore in OSD has a deep software stack for managing PM devices, which includes RocksDB, BlueFS, and PMDK. It leads to a high latency, aggravating the interaction issue. Similar to Octopus, the MDS is single-threaded which also impedes CephFS scalability improvement. Assise performs worse than Ethane. Even though Assise localizes data access with client-local PMs, its chain replication protocol incurs high remote write overheads. Specifically, every time a client creates a file, it needs to propagate this data update to all other remote nodes in sequence.

We evaluate two settings of file `stat`. In the first setting, file systems randomly access files and the experiment varies the file path length. Octopus achieves approximately 1 Mops/s regardless of the number of path component changes. The path resolution in Octopus is not POSIX-compliant. It hashes the whole path without any individual path component resolution. In contrast, Ethane faithfully resolves every component and it is still 1.8× faster than Octopus thanks to the parallel path walk design. Assise's throughput is close to that of Ethane. It is because there is no data replication during file `stat` and all data access happens on local devices. However, random file access causes numerous PM misses for Assise. Loading data from SSD-based storage delivers a similar or even longer latency than `RDMA_READ` latency.

The second setting uses a non-uniform access popularity. Both Octopus and CephFS configure one worker per SN. Therefore, their total throughput decreases dramatically when the file access becomes more skewed. Fortunately, Ethane has no such load imbalance concern. All `stat` requests are evenly distributed among cacheFS instances.

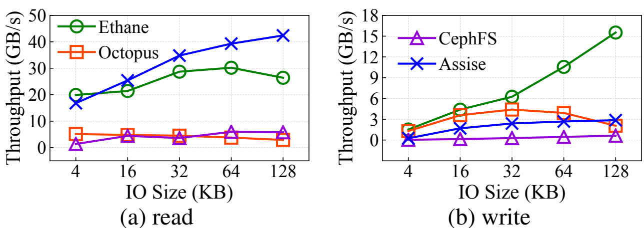


Figure 12: Fio Throughput

Data IO throughput. We use fio [4] to evaluate the data R/W performance. We spawn 32 clients per CN/SN. All client performs data reads to a shared file. We measure the data throughput with different IO sizes in Figure 12. CephFS performs worst. Data de-/serialization, message encapsulation, and extra data copies in its message-based RPC lead to low throughput. Its peak throughput only approaches 6 GB/s. Octopus has load imbalance issues. When the IO size increases, its total throughputs are bounded by a single node. Besides, Octopus achieves better performance than CephFS for small-sized IOs. Its client-active IO uses one-sided `RDMA_READ` to

reduce data transfer overheads.

For Ethane, both its file mapping and block reading in the data path fully exploit aggregated bandwidth provided by the remote PM pool. Its total throughput decreases when the IO size exceeds 64KB. Every time Ethane reads file data, it needs to initiate one extra remote read to check if there exist any new dependent logs. These additional data reads consume network and PM bandwidth. When running 32 threads with an IO size of 128KB, the bandwidth exhausts and the total throughput decreases. Assise read throughput is higher than Ethane. Every client in Assise caches the file in its local NVM. A local NVM read is 10× lower than a remote NVM read which accounts for their performance difference.

We spawn four clients per CN/SN for the write experiment and clients write a shared file. Analogously to the read experiment, when the IO size is large, the total file system performance of CephFS and Octopus are bottlenecked by the PM capability in an SN. They deliver a peak throughput of 0.63 GB/s and 4.4 GB/s, respectively. The write throughput in Assise is much worse than the read throughput. For a file write, it has to send the updated data to all remaining SNs. Such expensive data coherence protocol greatly degrades the write throughput. Ethane throughputs scale linearly thanks to its parallel data path design and reach a peak of 15.52 GB/s.

Cost efficiency. This experiment uses the video server workload from filebench [3]. This workload prepares a set of 1 GB video files. During experiments, a vidwriter writes new videos and fifteen clients read these video files with different IO sizes. This workload is read-intensive which stresses the file system IO performance.

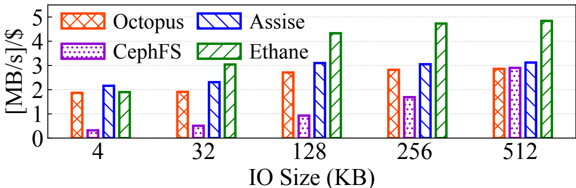


Figure 13: Performance-cost Evaluation

A single PM device only offers a peak of 6 GB/s bandwidth. For large IO sizes, file systems need to add more PM devices to serve the IO requests. Symmetric PM file systems add more monolithic SN while disaggregated PM file system plugs more PM modules into the MN. For different IO sizes, we choose the most cost-effective hardware configuration for each file system. This hardware configuration fitly supports running this workload. For example, for an IO size of 32 KB, CephFS requires two SNs. These two SNs contain four PM devices whose total bandwidth is sufficient for running the workload. One SN or three SNs is under- or over-provisioned.

Figure 13 shows the performance-cost efficiency. Ethane yields the highest throughput (i.e., MB/s) per dollar. The reasons are twofold. First, the disentangled data path design in Ethane brings better PM utilization than other file systems. Second, when PM resources become scarce due to increased IO size, Ethane only needs to add a new PM DIMM without purchasing an entire machine as other PM file systems. Two Intel DCPMMs cost $838 while an SN machine including two DCPMMs takes $3789. Disaggregating PM reduces significant monetary costs compared with symmetric PM.

5.4 Application Performance

Redis cluster. We evaluate the data persistence performance of a distributed key-value store: Redis cluster [9]. Redis cluster shards data and replicates Redis nodes to manage these shards. The Redis node supports two data persistence modes: (1) AOF, which persists every operation in a log and flushes logs periodically to disk; (2) RDB, which snapshots database states and checkpoints it to disk.

Experiments create sixteen shards and each client operates a shard by putting 100 million keys and executing the SAVE command to dump the database into a RDB file. We run the Redis cluster atop four file systems and measure both AOF throughput and ROB latency in Figure 14a. Ethane achieves 6.77%/17.98×/41.55% higher AOF throughputs than Octopus/CephFS/Assise. Replicated Redis is unfriendly to Assise for its expensive data coherence mechanism. CephFS is 10× slower than Ethane for its heavyweight software stack. When clients dump ten 10 GB RDB files, Ethane achieves 27.56%/8.21×/4.71× lower latency than others.

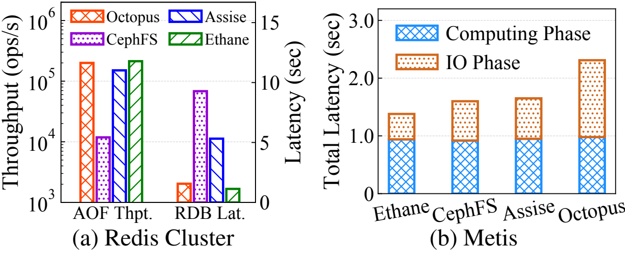


Figure 14: Application Performance

Metis. We run a multicore-optimized MapReduce application Metis [18]. We use Metis to run WordCount with a 16GB input file. We configure two SNs for symmetric PM file systems. Besides, We configure 0.5 CN and 1.5 MNs for Ethane. The half CN and MN is enabled by using one NUMA node of that machine. It ensures that the total costs of 2 SNs and 0.5 CN + 1.5 MNs are approximately equal. Two SNs have twice more cores than the half CN but they deliver similar computing phase latency. It suggests that the computing resource is over-provisioned for symmetric PM file systems. On the other side, Ethane has a shorter IO phase latency. Four PM devices is under-provisioned for symmetric PM file systems. Thanks to its elastic resource scaling, Ethane yields superior performance than others with the same hardware cost.

6 Related Works

Distributed file systems. For the past decades, DFSs play a critical role in large-scale data storage. Conventional file systems decouple metadata from data management, e.g.,

HDFS [63], PVFS [21], and Lustre [10]. It is a reasonable design for monolithic data center as a machine is capable of storing and manipulating file data. A line of research efforts have been devoted to improving capabilities in handling metadata requests and processing data IOs. CephFS [69] improves metadata scalability via namespace tree partitioning. GIGA+ [56] adopts a hash-based directory partitioning scheme. To avoid system-wide synchronization, GIGA+ disables client caching for high concurrency. IndexFS [59] and HopsFS [54] use NoSQL and relational databases for efficient small-sized metadata storage and indexing. HDFS [63] uses block replication to improve data availability and achieve aggregated IO throughput. QFS [55] reduces replication-incurred storage consumption via erasure coding.

PM-based file systems. For local file systems, researchers exploit PM characteristics to redesign various file system modules, such as namespace hierarchy [19], data IO [39], and journal mechanism [57, 71]. For distributed file systems, incorporating PM is flexible. BlueStore in Ceph [11] uses PM for OSD storage. However, it has a deep software stack for PM management, resulting in a long software latency. NVFS [36] proposes a PM-based write-ahead log design for HDFS. SINGULARFS [33] deploys all PM devices in one server machine. It only scales for billions of files due to the PM limitation of a single machine.

Octopus [49] and Orion [72] couple high-speed RDMA and PM. Octopus proposes a shared PM pool abstraction via unifying disjoint PM devices across multiple nodes. The weak node issue easily arises as it lacks efficient load balance mechanism. Orion [72] configures PMs in client machines, metadata servers, and data stores. Its scattered data introduces substantial node interactions during request processing. Assise [12] is a client-local PM file system. It achieves superior system performance by placing PM near client applications. However, this architecture design transfers PM expenses to users. These file systems extensively leverage PM's strength but they overlook PM's drawbacks.

Disaggregated PM system. Memory and storage disaggregation gains increased research interests recently [22, 27, 34, 35, 43, 46, 61, 62, 65, 75]. Resource disaggregation effectively overcomes the inherent storage capacity and cost deficiency of persistent memories. Moreover, commodity RDMA network [40] and forthcoming fast CXL protocols [45, 52] retain the latency advantage of PMs. This paper argues that disaggregated PMs provides an attractive and competitive solution towards future high-performance DFSs, and to the best of our knowledge, Ethane is the first file system that unleashes such hardware potentials with a novel asymmetric architecture and efficient functionality separation.

7 Conclusion

This paper revisits the PM usage in existing distributed file systems and reveals three correlated issues. To leverage PM performance strength as well as overcome its capacity and cost weakness, we propose a DPM-based file system Ethane. Ethane features an asymmetric file system architecture which decouples an FS into two planes running on distinct server nodes in DPM. Compared with modern PM-based distributed file systems, Ethane yields significant better performance for data-intensive applications with much lower monetary costs.

Acknowledgments

We thank the reviewers for their helpful feedback. This paper is supported by the Fundamental Research Funds for the Central Universities (Grant No. NS2024057) and the Natural Science Foundation of Jiangsu Province (Grant No. BK20220973). Baoliu Ye is the corresponding author.

References

[1] Apache Hadoop. https://hadoop.apache.org/, 2023.

[2] Apache ZooKeeper. https://zookeeper.apache.org/, 2023.

[3] Filebench. https://github.com/filebench/filebench, 2023.

[4] Flexible I/O Tester. https://github.com/axboe/fio, 2023.

[5] Intel PMWatch. https://github.com/intel/intel-pmwatch, 2023.

[6] libaco. https://github.com/hnes/libaco, 2023.

[7] MDTest. https://github.com/LLNL/mdtest, 2023.

[8] MPICH. https://www.mpich.org/, 2023.

[9] Scale with Redis Cluster. https://redis.io/docs/management/scaling/, 2023.

[10] The Lustre file system. https://www.lustre.org/, 2023.

[11] Abutalib Aghayev, Sage A. Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 years of Ceph Evolution. In *27th ACM Symposium on Operating Systems Principles, Huntsville, ON, Canada, October 27-30, 2019*, pages 353–369.

[12] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostic, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation, Virtual Event, November 4-6, 2020*, pages 1011–1027.

[13] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation, Virtual Event, November 4-6, 2020*, pages 617–632.

[14] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation, San Jose, CA, USA, April 25-27, 2012*, pages 1–14.

[15] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, Farmington, PA, USA, November 3-6, 2013*, pages 325–340.

[16] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of ACM*, 60(4):48–54, 2017.

[17] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, Virtual Event, November 4-6, 2020*, pages 753–768.

[18] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Symposium on Operating Systems Design and Implementation, October 4-6, 2010, Vancouver, BC, Canada*, pages 1–16.

[19] Miao Cai, Junru Shen, Bin Tang, Hao Huang, and Baoliu Ye. FlatFS: Flatten Hierarchical File System Namespace on Non-volatile Memories. In *2022 USENIX Annual Technical Conference, Carlsbad, CA, USA, July 11-13, 2022*, pages 899–914.

[20] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 24-27, 2020*, pages 209–223.

[21] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *4th Annual Linux Showcase & Conference, Atlanta, Georgia, USA, October 10-14, 2000*.

[22] Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, Haiqing Sun, Zechao Zhuang, Yuming Yang, Jie Xu, Liang Yin, Wenchao Zhou, and Sheng Wang. CloudJump: Optimizing Cloud Databases for Cloud Storages. *Proc. VLDB Endow.*, 15(12):3432–3444, 2022.

[23] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. Taming the Killer Microsecond. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, Fukuoka, Japan, October 20-24, 2018*, pages 627–640.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation, San Francisco, California, USA, December 6-8, 2004*, pages 137–150.

[25] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert van Renesse. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation, Santa Clara, CA, USA, February 25-27, 2020*, pages 325–338.

[26] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-space NVM File System. In *27th ACM Symposium on Operating Systems Principles, Huntsville, ON, Canada, October 27-30, 2019*, pages 478–493.

[27] Siying Dong, Shiva Shankar P., Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. Disaggregating RocksDB: A Production Experience. *Proc. ACM Manag. Data*, 1(2):1–24, 2023.

[28] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, Lombard, IL, USA, April 2-5, 2013*, pages 371–384.

[29] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. In

ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021, pages 440–456.

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43.

[31] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA, March 27-29, 2017*, pages 649–667.

[32] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.*, 14(4):626–639, 2020.

[33] Hao Guo, Youyou Lu, Wenhao Lv, Xiaojian Liao, Shaoxun Zeng, and Jiwu Shu. SingularFS: A Billion-Scale Distributed File System Using a Single Metadata Server. In *2023 USENIX Annual Technical Conference, Boston, MA, USA, July 10-12, 2023*, pages 915–928.

[34] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: a Hardware-software Co-designed Disaggregated Memory System. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February - 4 March, 2022*, pages 417–433.

[35] Haoyu Huang and Shahram Ghandeharizadeh. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 749–763.

[36] Nusrat Sharmin Islam, Md. Wasi-ur-Rahman, Xiaoyi Lu, and Dhabaleswar K. Panda. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing, Istanbul, Turkey, June 1-3, 2016*, pages 1–14.

[37] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 16-19, 2015*, pages 301–315.

[38] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 691–707.

[39] 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 *27th ACM Symposium on Operating Systems Principles, Huntsville, ON, Canada, October 27-30, 2019*, pages 494–508.

[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference, Denver, CO, USA, June 22-24, 2016*, pages 437–450.

[41] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event, Koblenz, Germany, October 26-29, 2021*, pages 756–771.

[42] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. Modernizing File System through In-Storage Indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation, July 14-16, 2021*, pages 75–92.

[43] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. *Proc. VLDB Endow.*, 15(13):4023–4037, 2022.

[44] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel Distributed Systems*, 31(1):94–110, 2020.

[45] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vancouver, BC, Canada, March 25-29, 2023*, pages 574–587.

[46] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A Scalable RDMA-oriented

Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 21-23, 2023*, pages 99–114.

[47] 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 *20th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 22-24, 2022*, pages 35–50.

[48] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, CA, USA, October 8-10, 2018*, pages 357–372.

[49] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference, Santa Clara, CA, USA, July 12-14, 2017*, pages 773–785.

[50] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 22-24, 2022*, pages 313–328.

[51] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 757–773.

[52] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vancouver, BC, Canada, March 25-29, 2023*, pages 742–755.

[53] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking File Mapping for Persistent Memory. In *19th USENIX Conference on File and Storage Technologies, February 23-25, 2021*, pages 97–111.

[54] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 89–104.

[55] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. A The Quantcast File System. *Proc. VLDB Endow.*, 6(11):1092–1101, 2013.

[56] Swapnil Patil and Garth A. Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 177–190.

[57] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Ninth Eurosys Conference on Computer Systems, Amsterdam, The Netherlands, April 13-16, 2014*, pages 1–15.

[58] Kai Ren and Garth A. Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 145–156.

[59] Kai Ren, Qing Zheng, Swapnil Patil, and Garth A. Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, November 16-21, 2014*, pages 237–248.

[60] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, Virtual Event, November 4-6, 2020*, pages 315–332.

[61] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87.

[62] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. Disaggregated RAID Storage in Modern Datacenters. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vancouver, BC, Canada, March 25-29, 2023*, pages 147–163.

[63] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10.

[64] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *56th Annual IEEE/ACM International Symposium on Microarchitecture, Toronto, ON, Canada, 28 October - 1 November, 2023*, pages 105–121.

[65] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference, July 15-17, 2020*, pages 33–48.

[66] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+ Tree Index on Disaggregated Memory. In *International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1033–1048.

[67] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA, March 27-29, 2017*, pages 35–49.

[68] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference, July 14-16, 2021*, pages 523–536.

[69] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *7th Symposium on Operating Systems Design and Implementation, November 6-8, 2006, Seattle, WA, USA*, pages 307–320.

[70] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, 6-12 November 2004, Pittsburgh, PA, USA*, pages 1–12.

[71] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 22-25, 2016*, pages 323–338.

[72] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies, Boston, MA, February 25-28, 2019*, pages 221–234.

[73] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182.

[74] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, Virtual Event, November 4-6, 2020*, pages 191–208.

[75] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, February 22-24, 2022*, pages 51–68.

[76] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling High-Performance and Secure Userspace NVM File Systems with the Trio Architecture. In *29th Symposium on Operating Systems Principles, Koblenz, Germany, October 23-26, 2023*, pages 150–165.

[77] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation, Virtual Event, November 4-6, 2020*, pages 1225–1240.

[78] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference, July 14-16, 2021*, pages 15–29.