Hydra: A Decentralized File System for Persistent Memory and RDMA Networks

Shengan Zheng, Jingyu Wang, Dongliang Xue, Jiwu Shu, *Fellow, IEEE*, and Linpeng Huang, *Senior Member, IEEE*

Abstract—Emerging byte-addressable persistent memory (PM) has the potential to disrupt the boundary between memory and storage. Combined with high-speed RDMA networks, distributed PM-based storage systems offer the opportunity to provide huge increases in storage performance by closely coupling PM and RDMA features. However, existing distributed file systems adopt the conventional centralized client-server architecture designed for traditional disks, leading to excessive access latency, limited scalability, and high recovery overhead. In this paper, we propose a fully decentralized PM-based file system, Hydra. By exploiting the performance advantages of local PM, Hydra leverages data access locality to achieve high performance. To accelerate file transmission among Hydra nodes, file metadata and data are decoupled and updated differentially through one-sided RDMA reads. Hydra also batches RDMA requests and classifies RPCs into synchronous and asynchronous types to minimize network overhead. Decentralization enables Hydra to tolerate node failures and achieve load balancing. Experimental results show that Hydra outperforms existing distributed file systems by a large margin, and shows good scalability on multi-threaded and parallel workloads.

Index Terms—Persistent memory, file system, RDMA, distributed system, decentralization

1 Introduction

Emerging persistent memory (PM), such as Intel Optane DC Persistent Memory Module (Optane DCPMM) [1], is blurring the line between memory and storage. Byte-addressable, non-volatile PM enables applications to directly access persistent data in the main memory. Compared with conventional storage devices (e.g., HDD or SSD), PM offers near-DRAM access latency and bandwidth. PM-aware storage systems promise to significantly improve application performance.

Recent advances in Remote Direct Memory Access (RDMA) technologies enable programmers to build efficient distributed storage systems that combine PM storage and RDMA networks. RDMA is a memory access technique that enables network interface cards (NICs) to directly access remote memory. Several PM-based distributed file systems [2], [3] have been proposed to exploit the benefits of PM and RDMA networks. Different from disk-based distributed file systems [4], [5], [6] that isolate file system and network layers for simplicity, these file systems install PM on server nodes to store metadata and data, and closely couple PM and RDMA features to achieve high performance and reliability. For metadata and data management, these file systems adopt the conventional centralized client-server architecture that uses main memory as the volatile cache for clients. Upon file access, file metadata and data are fetched from remote servers to the memory buffer on local clients to serve I/O requests.

However, conventional centralized client-server architecture falls short in leveraging the full potential of PM and RDMA networks in PM-based distributed file systems. First, compared with traditional disk-based distributed systems, the major performance bottleneck of PM-based distributed storage systems has shifted from storage to network. Contrary to the historical trend, network latency is expected to remain far above PM latency for the foreseeable future, due to propagation delay and network round trip [7]. Local PM I/O significantly reduces access latency than remote access via RDMA networks. Second, the centralized client-server architecture limits system scalability. Centralized servers simplify the coordination between clients at the cost of poor scalability. The problem becomes even worse when the servers are running on real Optane DCPMM, since its I/O performance does not scale well with thread count [8], [9], [10]. Third, client-side DRAM cache increases server monetary costs and introduces high recovery overhead upon power or system failure on client nodes. The per-GB cost of Optane DCPMM is only around 39% that of DRAM [11]. Moreover, DRAM-based client nodes must rebuild metadata and data cache from remote servers from scratch during recovery, which increases recovery overhead as well as server load.

The unique characteristics of PM and RDMA technologies provide the opportunity to interconnect the local PM-aware file systems on individual storage servers into a rack-scale

- *Shengan Zheng, Jingyu Wang, Dongliang Xue, and Linpeng Huang are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {shengan, wjy114, xuedongliang010, lphuang}@sjtu.edu.cn.*
- *Jiwu Shu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100190, China. E-mail: shujw@tsinghua.edu.cn.*

Manuscript received 3 October 2021; revised 28 April 2022; accepted 1 June 2022. Date of publication 9 June 2022; date of current version 23 August 2022.
This work was supported in part by the Natural Science Foundation of Shanghai under Grant 22ZR1435400 and in part by Shanghai Municipal Science and Technology Major Project under Grant 2021SHZDZX0102.
(Corresponding author: Linpeng Huang.)
Recommended for acceptance by D. Li.
Digital Object Identifier no. 10.1109/TPDS.2022.3180369

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

file system cluster, in which file metadata and data are disaggregated among cluster nodes and connected through high-speed RDMA networks. On each cluster node, file data is stored locally and durably, allowing applications to take advantage of the high performance and large capacity of local PM during runtime, and quickly recover after a system crash. However, using PM as persistent storage on application-side cluster nodes presents a host of challenges. The synchronization between stale and latest file copies on different cluster nodes should be efficient and scalable. The decentralized file system cluster should tolerate arbitrary node failure gracefully and balance the load among nodes during runtime.

We present Hydra, a decentralized file system designed to improve scalability and fault tolerance by taking advantage of directly accessible persistent memory and high-speed RDMA networks. Hydra is the first distributed persistent memory file system to systematically optimize for scalability throughout its design. Different from the traditional paradigm that strictly separates storage servers from clients, Hydra combines them into a set of PM-equipped nodes, and connects them with RDMA networks to form a decentralized file system cluster. Hydra leverages the data access locality by fully exploiting the performance advantages of local PM to minimize I/O latency. To accelerate file transmission among nodes, Hydra proposes *differential file update* scheme to transmit the file metadata and data differentially via one-sided RDMA reads from remote nodes. To achieve load balancing, file metadata and data are decoupled and replicated across Hydra nodes dynamically. Hydra also introduces RDMA request batching and RPC classification mechanisms to leverage the full potential of RDMA networks to minimize the overhead of file metadata and data transmission. Furthermore, Hydra supports online node addition and removal, providing high elasticity and flexibility.

The contributions of this paper include:

- We propose Hydra, a fully decentralized distributed file system that takes full advantage of data locality on local persistent memory to achieve high scalability, high availability, and crash consistency.
- We design a differential file update scheme to accelerate file transmission and node recovery by transmitting file metadata and data differentially through one-sided RDMA reads.
- We describe how Hydra tolerates arbitrary node failures and balances the load among cluster nodes.
- We implement and evaluate Hydra. Experimental results show that Hydra demonstrates good scalability on multi-threaded and parallel workloads, and significantly outperforms existing distributed file systems.

The remainder of the paper is organized as follows. Section 2 provides the background on PM-aware file systems and RDMA networks. Section 3 presents the design overview of Hydra. We describe the file transmission mechanism, cluster management, and RDMA optimization techniques in Sections 4, 5, and 6, respectively. The experimental results are presented in Section 7. Section 8 discusses related work, and Section 9 concludes the paper.

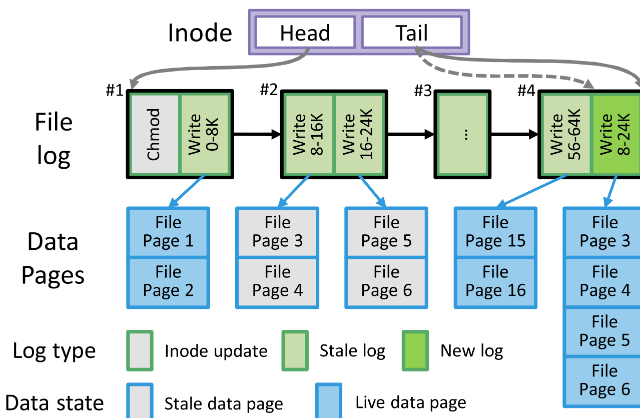


Fig. 1. *The file structure of NOVA and Hydra.* The file log contains inode update entries and file write entries. These file write entries contain pointers to data pages.

2 Background And Motivation

Hydra is a distributed file system designed for PM and RDMA networks. This section provides background on PM, the NOVA file system that Hydra is based on, and RDMA technology. We conclude by motivating the need for a new distributed PM file system that is fully decentralized.

2.1 PM and PM-Aware File Systems

Persistent memories such as STT-RAM, ReRAM, and PCM provide data persistency, byte-addressability, as well as access latency and bandwidth within an order of magnitude of DRAM. Persistent memories are directly attached to the main memory bus alongside DRAM, and can be accessed via a load/store interface. Intel Optane DCPMM [1] is the first commercially available persistent memory DIMM.

The high performance and non-volatility of PM have attracted extensive research efforts on local PM-aware file systems in the last decade [12], [13], [14], [15], [16], [17], [18], [19], [20]. Among them, NOVA [17], [21] is a state-of-the-art PM-based file system designed to maximize performance while providing strong consistency guarantees. NOVA uses per-CPU free lists, journals, and inode tables to ensure good multi-core scalability. For each file, NOVA maintains a separate log in PM, which consists of a singly linked list of 4 KB log pages. The relationship between the inode, its log, and its data pages is illustrated in Fig. 1. The tail pointer in the inode points to the latest committed entry in the log. For each file update, NOVA creates the corresponding log entry in its log, and atomically updates the log tail pointer in PM. To accelerate file access, NOVA maintains a radix tree in DRAM that maps file offsets to data page addresses in PM.

Since NOVA fully exploits the performance benefits of PM and the multi-core scalability of CPU, and its log-structured file design naturally fits Hydra's differential file update scheme for its linearizability, we implement Hydra based on NOVA. Specifically, Hydra inherits the design of log-structured file layout and scalable memory allocators from NOVA. In Hydra, the head and tail pointers are also used to maintain the file logs connected through linked lists. Stale files can be synchronized with their latest versions by transmitting and replaying the differential file logs. Nevertheless, we also make necessary adjustments to the content of log

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

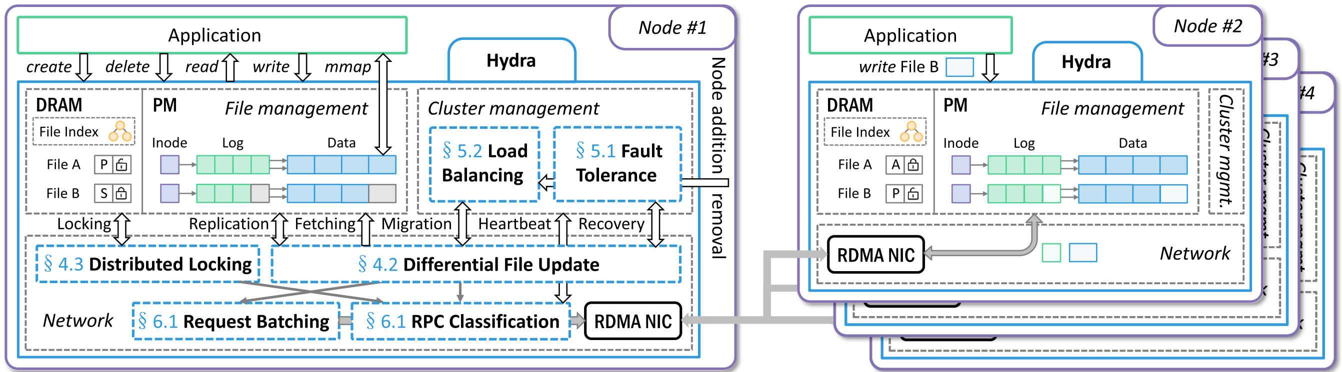


Fig. 2. *The architecture of Hydra.* Hydra fully exploits the access locality of local PM and efficient file transmission via differential file update to provide high scalability and availability to the applications.

entries to enable Hydra to manage the log and data pages across distributed cluster nodes. Hydra also maintains an inode table, journal, and PM free page list at each CPU to avoid global locking. This helps Hydra to avoid scalability bottlenecks on each cluster node.

2.2 Remote Direct Memory Access

RDMA provides low-latency and high-bandwidth remote memory access by directly accessing memory from remote servers. In the Reliable Connection (RC) mode, nodes are connected with a pair of send/receive queues (queue pairs, QPs). Applications initiate RDMA requests by placing work queue entries (WQEs) in the send queue. After completing an RDMA request, the NIC signals the completion by posting a completion queue entry (CQE) to the completion queue (CQ). The applications can poll for the completion from the CQ to receive notification that the requests have been completed successfully.

RDMA supports read/write operations (one-sided verbs) as well as send/receive operations (two-sided verbs) between two nodes. One-sided verbs deliver higher throughput by directly accessing the pre-registered memory region exposed by the remote host, without notifying the host CPU. For two-sided verbs, the remote CPU is involved in handling the RDMA requests. In addition, RDMA provides atomic verbs such as compare-and-swap (`CAS`) and fetch-and-add (`FAA`) to atomically update 64-bit data in remote memory.

Existing RDMA-based RPC requests are sent by either one-sided [22], [23] or two-sided [24], [25], [26], [27], [28], [29] verbs. Requests sent through one-sided verbs are written to a dedicated message buffer on the server. The server threads busy checking the message buffer for new request arrival. Such design provides low latency RPC processing at the cost of high server CPU overhead, which does not fit a scalable decentralized system. Hydra adopts two-sided verbs for its RPC implementation to eliminate the cost of busy message checking at the receiver side, so as to balance the load among cluster nodes.

2.3 Challenges in Distributed PM File Systems

Emerging high-performance PM and RDMA technologies are both appealing building blocks for high-performance distributed storage systems. Incorporating these hardware technologies in distributed file systems poses new challenges in terms of (1) system scalability, (2) access latency, and (3) fault tolerance.

System Scalability. In the conventional client-server architecture, the centralized metadata nodes are used to handle the metadata access throughout the entire cluster. Even when employing techniques like replication and partitioning, the metadata nodes are still vulnerable to skewed file accesses from massive concurrent accesses. Furthermore, the scalability of RDMA NICs declines with the number of active QPs due to their limited memory [30], leaving the file system unable to provide scalable metadata service under heavy workloads.

Access Latency. Compared with the components of traditional distributed file systems (such as hard disks and Ethernet), PM and RDMA networks provide substantially higher performance, which exposes the latency bottlenecks within the storage stack. At the client side, excessive data copies in user, kernel, and network buffers drastically increase the latency of remote data access. At the server side, traditional RPC handlers introduce substantial CPU overhead during request processing. Moreover, file data is often transferred at a coarse granularity, while PM and RDMA networks support byte-addressable data access, resulting in severe write amplification and increased data access latency.

Fault Tolerance. Distributed file systems should be capable of tolerating system failures while maintaining high availability for both file data and metadata. The potential failures of distributed file systems include cluster node failures and network partitions. However, distributed file systems that adopt the centralized client-server architecture are vulnerable to both. Although some file systems improve availability through replication approaches, they still suffer from high file access latency and suspended file service during fail-over.

Distributed PM file systems demand new methods to improve system scalability, minimize access latency, and tolerate failures gracefully. As such, we present Hydra, a fully decentralized PM file system that fully utilizes the unique characteristics of persistent memory and high-speed RDMA networks.

3 Design Overview

Hydra is built for a cluster of servers that are equipped with PM and RDMA networks. Fig. 2 summarizes its high-level design components and presents an overview of the Hydra

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

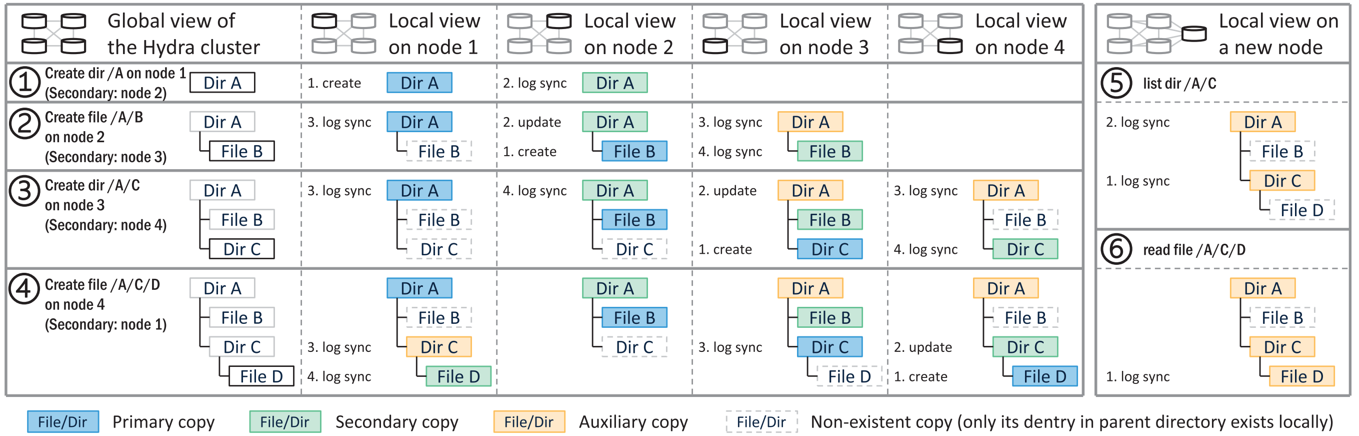


Fig. 3. *The global and local views of the Hydra cluster.* After the directories and files have been created on a Hydra cluster (step ①-④), each node contains a fraction of the entire file set, but all the files in the cluster are accessible to every node. For newly connected nodes, Hydra creates auxiliary copies to facilitate directory and file accesses (step ⑤ and ⑥).

system. In Hydra, a typical I/O workflow of file update consists of four stages: (1) Permission checking stage. Hydra acquires the write permission through a lightweight distributed locking module. The decentralized file locks are transferred via low-overhead RDMA atomic operations (Section 4.3). (2) File synchronization stage. Each Hydra node manages its local files in a log-structured layout. To fetch the latest file metadata and data or handle replication requests from remote nodes, Hydra utilizes a *differential file update* scheme (Section 4.2) to transmit differential file logs efficiently through one-sided RDMA reads. (3) Local I/O stage. Hydra serves I/O requests with the latest file data on the local node, taking the advantage of data locality to achieve high performance. After file updates, new log entries are appended to the file log. (4) Data transmission stage. Hydra issues RPC requests to remote nodes to commit the update. To reduce network overhead and provide scalable performance, Hydra employs RDMA request batching and RPC classification mechanisms (Section 6.1).

In addition, Hydra achieves fast recovery after a system crash (Section 5.1) and efficient migration during load balancing (Section 5.2) by exploiting the differential file update scheme. Hydra also exposes a normal POSIX interface on each cluster node and supports dynamically adding or removing cluster nodes during runtime, providing high elasticity and flexibility to applications. Overall, Hydra achieves the following design goals:

- *Scalable performance:* Multi-core and multi-node scalability are essential for Hydra to effectively harness the performance advantages of PM and RDMA networks. Hydra leverages data locality by placing file data in application-side PM to improve overall performance and achieves multi-node scalability by adopting RDMA request batching and RPC classification mechanisms.
- *File transmission with low overhead:* Hydra leverages the linearizability of file logs to implement differential file update for synchronizing stale files among cluster nodes. Files are synchronized differentially among nodes via one-sided RDMA reads, which accelerates file transmission and eliminates remote CPU bottleneck.
- *High reliability and availability:* File metadata and data are dynamically replicated across all decentralized Hydra nodes, enabling Hydra to tolerate node failures without data loss. In the event of a node failure, applications can fail-over to any other node in the cluster.
- *Lightweight coherency mechanism:* Since there is no central metadata server to arbitrate concurrent file updates, we employ lock tokens to grant file-granularity write permissions to coordinate simultaneous file accesses in a Hydra cluster. The tokens are exchanged simply through local/RDMA compare-and-swap (CAS) operations, minimizing coherence overhead.
- *High elasticity:* Hydra supports dynamically adding or removing cluster nodes during runtime, providing high elasticity and flexibility to applications.

To achieve these design goals, unlike conventional distributed file systems that strictly differentiate cluster management nodes, metadata server nodes, data server nodes, and client nodes, Hydra combines their functionalities into a single decentralized file system module. Upon file access, Hydra exploits locality by performing file I/O directly on the local PM. If the local file is stale or non-existent, Hydra will update the file to the latest version in the cluster differentially. As for replication, each file in Hydra has at least two copies (a primary copy and a secondary copy) of metadata and data that are distributed among cluster nodes by default. This enables Hydra to tolerate arbitrary single node failure without data loss.

4 File Transmission

In this section, we describe how Hydra significantly accelerates file transmission among cluster nodes. We first illustrate how Hydra updates file metadata and data differentially. Then, we discuss how Hydra maintains coherency among the cluster nodes. Finally, we summarize the design of Hydra by walking through some typical directory and file operation examples.

4.1 Distributed File Management

As shown in Fig. 3, Hydra has no centralized management node. Instead, from each file's perspective, Hydra has three types of file copies distributed among the cluster nodes.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

Primary copy is the authoritative copy of the file. Each file in Hydra has one primary copy, which is synchronously updated. The primary copy has the longest metadata log of the file, as well as a complete copy of file data. The node that creates the file is designated as its primary node by default. The identity of the primary node is stored in the inode and it may change due to migration (during load balancing) or primary election (upon node failure). In these cases, Hydra designates another node as primary, and broadcasts the message to all relevant nodes through RPCs.

Secondary copy is a replicated copy of the file. The nodes that hold the secondary copies are determined by a variant of the *power of two choices* algorithm during file creation (Section 5.2). The secondary copies are asynchronously updated, unless `fsync` is called to synchronize the file. The number of secondary copies is set by the user. By default, there is one secondary copy of each file in the Hydra cluster.

Auxiliary copy is an extra copy of a file. When a file is replicated to a Hydra node, the file itself and its parent directory must be synchronized with other nodes in order to maintain the directory tree structure of the local file system view. If the local node does not currently possess either the primary or secondary copy of the file/directory, Hydra will create an auxiliary copy of it. This ensures that each Hydra node is able to retrieve all its local files via directory tree walk, and runs like a local file system even when all other nodes in the Hydra cluster are lost. The auxiliary copies of files are subject to eviction in an LRU manner when the PM usage of the node is high. These copies will be retrieved from the primary/secondary nodes upon next file access.

Hydra provides high availability to applications. As shown in Fig. 3 step ④, files and their replicas are distributed among cluster nodes in Hydra. Each node only contains a fraction of the entire file set, but all the files in the Hydra cluster are accessible to every node. Stale or non-existent auxiliary copies of files on the local node are only synchronized with the latest version when they are accessed. From each file's perspective, we refer to the node that holds the primary/secondary copy of the file as the *primary/secondary node* of the file.

Hydra nodes are identified with unique 8-bit global IDs, which are generated by hashing the RDMA addresses of the nodes by default. All the nodes in the Hydra cluster periodically communicate with each other via heartbeat messages. The heartbeat messages contain the current CPU, network, and PM usages, which are used for load balancing.

4.2 Differential File Update

Hydra uses differential file update scheme (`diff-update`) to update stale (or non-existent) files/directories on the local node with their latest versions on a remote node. `Diff-update` enables any Hydra node to fetch the latest version of any file from other nodes in the cluster. As shown in Fig. 2, `Diff-update` is widely used in Hydra for transmitting directories and files in various scenarios. Hence, the local and remote transmission overhead of `diff-update` is crucial for the overall performance and scalability of the Hydra cluster. We go through the steps involved during `diff-update` in this section.

File Version Verification. Before pulling metadata and data from a remote node, Hydra first verifies whether the local

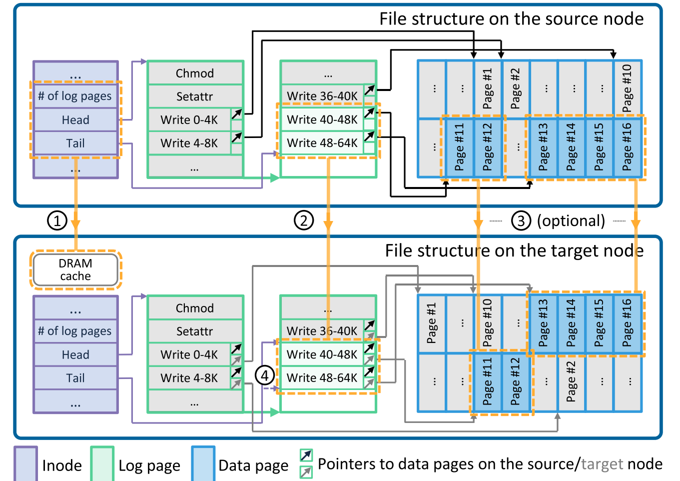


Fig. 4. *Differential file update.* The local (target) node synchronizes with the remote (source) node to update the local stale file copy to the latest version differentially.

version of a file (or a directory) is up-to-date or stale through local and remote verification. First, Hydra checks whether the local node is either the primary node or the token-holder (Section 4.3) of the file. Since any committed modifications are mirrored in the primary node, and only the token-holder can modify the file, the file content of either the token-holder or the primary node is up-to-date.

If local verification fails, Hydra turns to remote verification. Hydra issues a small RDMA read to the file metadata of the primary file copy to retrieve the number of log pages and the head and tail pointers (Fig. 4 step ①). The head pointer is used to indicate the starting point of the file log, while the tail pointer is used to calculate the *log length* of the file. If the local log length matches the remote one, the local file is up-to-date. This is because Hydra coordinates file locks through lock tokens (discussed in Section 4.3). For any file, only one of the nodes is able to modify it and increase its log length. As a result, the latest file version is the file copy of the node with the longest log length. If the local file is up-to-date, the whole synchronization process takes only one network roundtrip. Otherwise, the local file has become obsolete. Hydra initiates the process of differential metadata and data updates.

In Hydra, the *log length* is calculated by comparing the number of log pages and the offset of the tail pointer. As illustrated in Fig. 1, the number of log pages is the logical number of the last log page. Therefore, the logical log length is essentially equivalent to the version number of each file in Hydra. When a new log page is allocated to the file, the number of log pages is incremented by one. During garbage collection, old log pages that only contain invalid logs are reclaimed. However, the logical number of log pages will not decrease. If the numbers of the local and remote log pages match, Hydra compares the in-page offset of the tail pointers for an accurate comparison of log length.

Metadata Update. Hydra inherits the log-structured file design of NOVA to organize file metadata and data. Metadata update in Hydra is implemented by transmitting differential file logs from the source node to the target node. Since the source node with the latest version of the file is usually the bottleneck during file replication, we choose

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

RDMA reads over RDMA writes for metadata and data updates. For the source node, not only do one-sided RDMA reads bypass its CPU, serving inbound RDMA reads also imposes less overhead than issuing outbound RDMA writes [31].

As shown in Fig. 4, after remote verification ①, the addresses of the begin and end log pages are cached locally. Combining with the number of pages on both nodes, Hydra is able to calculate the corresponding log address on the remote node that matches the current log tail on the local node. Then, the local node pulls the differential logs after its current log tail through an RDMA read request ②, and updates the log tail accordingly. Hydra recursively pulls the subsequent log pages, if any, using the *next-page* pointer in the current log page, until the local log length matches the remote one.

When the log pages have been fetched, Hydra replays the logs on the local node to update metadata. For instance, file-mode-change logs are applied directly, whereas file write logs are replayed by updating the file radix tree with local/remote data page pointers in DRAM.

During metadata update, file data is optionally updated depending on the request type. For instance, *file synchronization* focuses on data integrity, and calls `diff-update` with data update. Meanwhile, *file open* pursues fast log replication, therefore omits data update. In this case, local data logs still contain valid pointers to the remote data on other Hydra nodes (i.e., the black arrows in Fig. 4), which can be used to retrieve file data upon future access. Optional data update lowers the latency of `diff-update` by postponing data transmission.

Data Update. If the local node requests data update along with metadata update, the data pages will be fetched through batched RDMA reads (Section 6.1) to reduce transmission overhead ③. During the replay of file write logs, Hydra allocates data pages on local PM and asynchronously issues the corresponding RDMA read requests to fetch the data pages from the remote node. In Hydra, each file write log contains eight global data pointer slots. Each global pointer encapsulates a node ID (8 bytes) and the corresponding data page address (56 bytes) of a current primary/secondary node that holds the data. For data update, Hydra scans the pointer slots for valid addresses, and selects the node with the minimal network load to transmit data. During data transmission, Hydra decapsulates the global node ID and the logical address of the remote page, and performs a remote memory access via RDMA. After that, Hydra updates the local data page addresses in the logs (i.e., the grey arrows in Fig. 4). When all the newly fetched logs have been replayed, Hydra pulls the CQ for the completion of the batched RDMA read requests of file data pages.

Directory Update. During `diff-update`, directories are treated as log-only files with no data. File creation/deletion logs are replayed by adding/removing the corresponding dentries to/from its DRAM cache. Note that for the file creation logs in the directory, the newly created sub-files are not actually created on the local node by the log replay. Only their corresponding dentries are inserted into the directory. The newly created sub-files are locally created and synchronized only when they are accessed.

Hydra adopts a *lazy file synchronization* approach to only update the stale directories on the bottom-up directory path of the accessed files on-demand. As illustrated in Fig. 3 step ④, to create file *D* in directory *C* on node 4, *C* is synchronized with its primary node (i.e., node 3). Since directory *C* is up-to-date, the recursive synchronization is completed. Then, file *D* can be created locally on node 4. Note that directory *A* and file *B* are not involved in the creation of file *D*.

Through directory tree walk and on-demand recursive directory updates, the lazy file synchronization approach also guarantees that all the files in the Hydra cluster are accessible to every Hydra node. By decoupling directories and their sub-files, Hydra eliminates significant communication overhead of exchanging redundant file modification and directory structure information among nodes.

Consistency Guarantee. After the metadata and data have been synchronized, Hydra updates the number of log pages and the log tail pointer at the file's inode in local persistent memory to complete `diff-update` (Fig. 4 step ④). Since the log tail pointer update is an atomic operation, the consistency of `diff-update` is guaranteed.

File Replication. `Diff-update` facilitates efficient file replication in Hydra. To initiate file replication, the primary node sends asynchronous RPC requests (Section 6.1) to secondary nodes. Then, it handles inbound RDMA read requests of `diff-update` issued by secondary nodes concurrently and asynchronously. (Unless the application calls `fsync`, which forces file replications to complete synchronously.) When `diff-update` completes, the secondary nodes will send asynchronous RPC requests to the primary node to update the global pointer slots. After that, other nodes can fetch file data from either the primary node or secondary nodes. In contrast, conventional primary replication uses synchronous outbound RDMA writes with serialized chain-replication approach, leading to higher latency and lower scalability.

The `diff-update` approach offers three major advantages: First, transmitting differential file logs significantly reduces transmission overhead. `Diff-update` enables Hydra to coalesce duplicated changes in the packed file logs to improve transmission efficiency. Second, handling one-sided RDMA reads issued from target nodes eliminates CPU bottlenecks at the source node, providing higher scalability. Third, by decoupling file metadata (logs) and data, files can be synchronized by only updating metadata to the latest version, reducing the critical-path latency of file accesses.

4.3 Decentralized Distributed Locking

Since most datacenter workloads are read-mostly [32], Hydra supports multiple readers and single writer among all the Hydra nodes in the cluster. Hydra coordinates file locks across nodes with lock tokens. Each file has only one global lock token to grant write permission to one of the file copies. The lock token is managed by the primary node of the file.

We use a lightweight token-based locking mechanism with atomic operations to minimize token transfer overhead. Initially, the value of the file's lock token field on the primary node is set to zero. To acquire the lock token, the primary/non-primary node issues an local/RDMA compare-and-swap (`CAS`) request to the lock token on the primary node. If

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

the lock token is successfully acquired, the value of the lock token on the primary node is swapped with the unique 8-bit global ID of the node. When the lock token is released, its value will be reset to zero.

In order to reduce the quantity of network communications between nodes, we employ a lazy token transfer approach. In Hydra, the primary node does not actively revoke the lock token. Instead, when a write operation on the local node fails to acquire the lock token, the `CAS` operation returns the node ID of the current token-holder. In this case, the local node determines whether the token-holder is alive through recent heartbeat messages. If the token-holder node is alive, the local node will send a synchronous RPC message to that node, and then retry the `CAS` request until the lock is released. Upon receiving the message, the token-holder node will reset the lock token value on the primary node back to zero once its pending writes are completed. If the token-holder node or the primary node is down, the local node will initiate a *primary election* (Section 5.1) to recover the distributed lock.

The token-holder node differs from the non-token-holder nodes in two aspects. First, both read and write permissions of the file on the token-holder node can be granted by its read-write semaphore, whereas only read permissions can be granted by the non-token-holder nodes. Second, in order to avoid inconsistent logs, garbage collection to the file can only be performed on the token-holder node. The other nodes will synchronize file logs and data from the latest, garbage-collected file copy.

4.4 Directory and File Operations

Applications access the directories and files in Hydra via the POSIX interface. In this section, we show how Hydra handles directory operations to a file (*file*) in a directory (*dir*) as well as file operations to *file*. Note that *file* can be either a sub-file or a sub-directory in directory operation examples.

File Creation. File creation involves appending a new directory entry (dentry) to *dir*, as well as creating the metadata of *file*. In order to append a new dentry, the local Hydra node first obtains the permission for inserting the new dentry to *dir* by acquiring the lock token of *dir*, and then calls `diff-update` to synchronize *dir* with its primary node.

After *file* has been created on the local node, Hydra issues an RPC request to the primary node of *dir* to notify the newly inserted dentry, unless the local node itself is the primary node of *dir*. The primary node synchronizes *dir* with the local node, replays the new log that creates *file*, and then replies to the local node. After that, the primary node issues asynchronous RPC requests (Section 6.1) to update *dir* on the secondary nodes.

File Deletion. File deletion can be broken down into two steps: `unlink` (remove the dentry of *file* from *dir*) and `evict` (free the storage space that *file* occupies). `Unlink` is implemented similar to file creation. We use synchronous file update requests to inform the primary and secondary nodes of *dir* to remove the corresponding dentry. `Evict`, however, does not have to be completed synchronously. Hydra only needs to issue asynchronous RPC requests to the primary and secondary nodes to free the resources of *file*.

Directory Lookup. To lookup a dentry within *dir*, Hydra first synchronizes *dir* with its primary copy via `diff-update`, then conducts the lookup of *file*. If the directory is stale or non-existent before the lookup, Hydra will synchronize its parent directories recursively from the bottom up.

File Open. If the local node currently holds the lock token of *file*, then *file* is opened the same way as in a local file system. For non-token-holder nodes, Hydra first acquires the corresponding lock, and then issues a `diff-update` request to the primary node to update the local file logs. When `diff-update` is finished, Hydra wakes up a background thread to replicate file data to the local node asynchronously.

File Read. If the target data has been replicated to the local node, Hydra handles the read requests locally. Otherwise, Hydra fetches the requested data from the primary node with the pointers in the corresponding local file log to serve the read request. A local copy of the data is created simultaneously so that future reads become local. These auxiliary file copies are subject to eviction when the local PM usage is high.

File Write. File writes are directed to local PM to reduce latency. The consistency of file writes is guaranteed by the atomic log tail pointer update on the local file. When a file write completes, an asynchronous RPC request to the primary node is issued. The proxy threads (Section 6.1) on the primary node will pull the updated data on the local node with `diff-update`, and then propagate the updates to the secondary nodes asynchronously.

File Synchronization. When an application calls an `fsync` to a file, a synchronous RPC request is sent to each primary and secondary node to update the file log and data with the local node using `diff-update`.

Memory Mapping. To handle the `mmap` requests, Hydra maps the pages on persistent memory into the application's address space. The file is synchronized with the primary copy when the application unmaps the pages.

5 Cluster Management

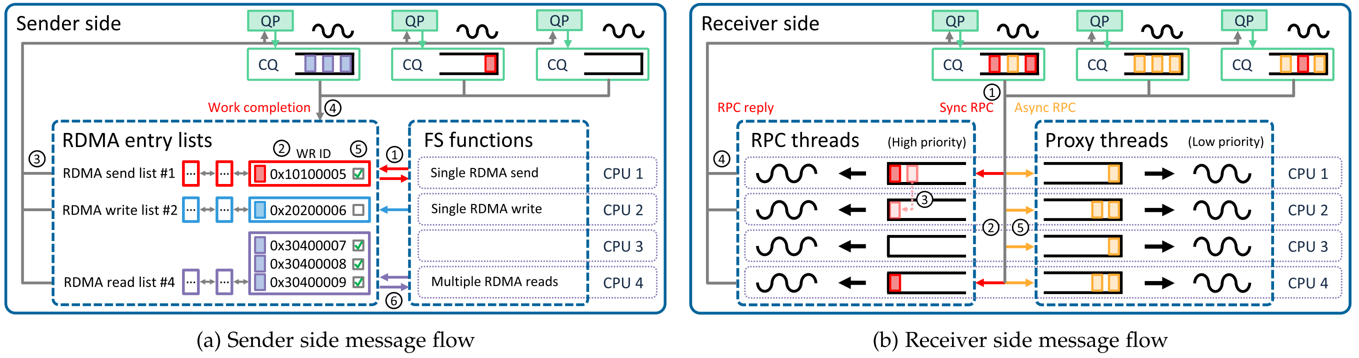
As a Hydra cluster scales to a large number of nodes or serves applications for a long period of time, it becomes increasingly unlikely that all the nodes are working correctly and all the file accesses are balanced. In this section, we describe how Hydra handles node failures gracefully and achieves load balancing during runtime.

5.1 Fault Tolerance

Hydra provides high availability to the applications by tolerating arbitrary node failures during runtime. In the scenario described in Fig. 3 step ④, since there are two copies of each file, arbitrary single node failure can be tolerated by Hydra during runtime with the guarantee that all the files can be retrieved. For the Hydra cluster, by setting the replication factor to n, concurrent failures of any $n-1$ nodes can be tolerated without any data loss. Decentralization enables Hydra to isolate failed nodes from the rest of the cluster nodes.

Node Failure. If a file is accessed by a non-primary node while its primary node is down, the non-primary node will initiate a *primary election* procedure to elect a new primary node for the file. The primary election is implemented based on the Paxos consensus protocol [33]. The node with the

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.



(a) Sender side message flow (b) Receiver side message flow

Fig. 5. *The sender and receiver side message flow of a Hydra node.* Hydra batches RDMA requests at the sender side, and classifies RPC requests into synchronous and asynchronous types at the receiver side.

longest file log will be elected as the new primary node. If none of the living nodes contains another file copy, the initiator becomes the primary node. Once the primary node has been elected, the new primary node will send RPC requests to the secondary nodes to make sure all other file copies in the Hydra cluster are up-to-date.

Node Recovery. If the crashed node is recoverable, it will rejoin the Hydra cluster after reboot. After that, it updates the root directory to the latest version, and then continues file service. Outdated files are synchronized with the latest version when they are accessed by directory tree walk from the root. Users may also run `fsck` to force update all the local files to the latest version. Note that since the local file logs are still valid, we only need to pull the newly-generated file logs that are committed during the downtime of the node through `diff-update`, instead of replicating the whole file. This significantly improves the efficiency of node recovery.

5.2 Load Balancing

Hydra utilizes a variant of the *power of two choices* algorithm [34] to achieve load balancing. The primary node uses two consistent hashing [35] functions to choose the secondary node of a file by its inode number. Between the two candidate nodes, the one with the lower PM usage is chosen as the secondary node. If Hydra is configured with more than one secondary node, then the other node is the second choice of the secondary node, and the rest of the secondary nodes are chosen in the ascending order of their PM usage.

Data Migration. Hydra adopts the hierarchical file system design that decouples directories and their sub-file/sub-directories completely, so as to balance the load among Hydra nodes. For Hydra nodes with high PM usage, file data is migrated to the node with the lowest PM usage.

Node Addition. New servers can be introduced into the Hydra cluster flexibly. They will join the consistent hashing ring with unique global IDs, so that new files created on other Hydra nodes can designate them as the primary or secondary nodes.

As a fully decentralized file system, Hydra strives to be highly scalable and expandable throughout its design. At the cluster-level, Hydra supports dynamically adding or removing cluster nodes during runtime, as well as tolerating arbitrary node failures. This provides high elasticity and flexibility to the cluster configurations, enabling online expansion of the distributed file system. At the node-level, Hydra organizes files in a fully decentralized manner that eliminates the bottleneck from centralized metadata management, providing high expandability. Coherency and consistency among nodes are also guaranteed through a lightweight decentralized distributed locking approach, minimizing the overhead of lock exchange in large-scale Hydra clusters. At the file-level, Hydra synchronizes file updates through differential file update scheme via one-sided RDMA reads. During file synchronization, the remote CPU is bypassed, allowing for more concurrent accesses from applications on different Hydra nodes. At the data-transmission-level, Hydra achieves high scalability through RDMA request batching and RPC classification mechanisms to minimize network overhead and facilitate high concurrent access from other Hydra nodes. We discuss these two mechanisms in the next section.

6 Implementation

In this section, we first demonstrate our approach to batch RDMA requests and classify RPCs to achieve scalable RDMA communication among Hydra nodes. Then, we describe how Hydra maintains data consistency on each node.

6.1 Scalable RDMA Communication

Compared with traditional storage and network devices, PM and RDMA technologies offer significantly lower latency, higher bandwidth, and higher scalability, which exposes the latency and scalability bottlenecks in the storage stack. We propose RDMA request batching and RPC classification mechanisms for network communication in Hydra, aiming to achieve scalable connection management and request handling by fully utilizing the benefits of technological advances in high-speed PM and RDMA networks.

RDMA Request Batching. At the sender side of each Hydra node (Fig. 5a), Hydra issues RDMA requests with per-CPU RDMA entry lists. This avoids request contention and allows for parallel request issuing and handling. When RDMA requests are issued, an RDMA entry that contains the WR is created and appended to the corresponding entry list ①. The first 12 bits of the 64-bit `wr_id` are reserved to indicate the RDMA operation type (4 bits) and the CPU ID (8 bits) ②. Hydra posts the RDMA verbs to QP, and modifies the RDMA request state to *posted* ③. When the RDMA operation is completed, the CQ handler will pull the CQE from CQ ④, trace the RDMA entry with `wr_id`, and update the RDMA request state to *completed* ⑤.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

Hydra is capable of coalescing multiple RDMA requests into a single entry to enable RDMA request batching. Large file I/O requests from applications or file data replication among Hydra nodes may issue massive asynchronous RDMA requests. Batching these requests significantly reduces the overall transmission latency. The corresponding file system functions will be blocked until all the RDMA requests within an RDMA batched entry are completed ⑥.

RPC Classification. At the receiver side of each Hydra node (Fig. 5b), the RPC requests are classified into two categories: synchronous and asynchronous. Synchronous RPC requests, such as file replication during `fsync`, are handled by the RPC threads with high priority to reduce latency. When an RPC request is received by a CQ handler ①, the request will be immediately dispatched to the wait-list of the corresponding per-CPU RPC thread according to its `wr_id` ②.

In order to achieve scalable request processing, Hydra amortizes the overhead of CPUs over multiple RDMA requests through work-stealing. If the designated RPC thread is currently busy handling other RPC requests, the new RPC request will traverse the wait-lists of other RPC threads and select the CPU with the lowest load ③. After the request has been processed, the receiver replies to the sender ④. Asynchronous RPC requests, such as background file migration, are handled by per-CPU proxy threads. These proxy threads act as the proxies of other nodes to perform tasks asynchronously ⑤. The asynchronous RPCs are handled with low priority to minimize the performance impact.

6.2 Data Persistence

In Hydra, the persistence of file updates, including local file writes and inbound RDMA reads during file transmission, is guaranteed via the combination of `clwb` and `sfence` instructions to flush the data from the processor cache to PM. For RDMA reads, Intel Direct Data I/O (DDIO) technology supports direct communication between RDMA networks and the last level cache (LLC) [36]. Since DDIO significantly improves the performance of data service [7], [37], [38], we enable DDIO for evaluation in our work. Hence, the destination of inbound RDMA reads becomes the local LLC, which may evict the cacheline in any order. To ensure the durability of file transmission, Hydra synchronously flushes the transmitted data to PM.

7 Evaluation

In this section, we evaluate the performance of Hydra against existing distributed file systems that support PM and RDMA, as well as local PM-aware file systems. Our evaluation answers the following questions:

- How do different Hydra replication configurations affect its I/O performance? (Section 7.2)
- How does Hydra perform against other distributed file systems on multi-threaded workloads? (Section 7.3)
- How large is the performance gap between local and remote file access of Hydra? (Section 7.4)
- How does Hydra scale over parallel workloads on multiple nodes? (Section 7.5)
- How do metadata operations perform on Hydra? (Section 7.6)
- How quickly can Hydra recover from failures? (Section 7.7)

7.1 Experimental Setup

We run Hydra on a 4-node cluster. Each node is equipped with two Intel Xeon Gold 6240 CPUs (running at 2.6 GHz with 36 physical cores), 384 GB DDR4 DRAM, 12 Optane DCPMMs (128 GB per module, 1.5 TB in total). Each cluster node has a Mellanox ConnectX-5 InfiniBand network adapter that connects to an InfiniBand switch.

We compare Hydra with five distributed file systems on the same cluster: CephFS [4], GlusterFS [6], NFS [39], Octopus [2], and Assise [11]. For a fair comparison, the clients and servers of all these distributed file systems are connected with the same RDMA networks and run on all four PM-equipped cluster nodes (except NFS which only has one server node). For disk-based distributed file systems, we use PM on the server nodes as the storage device for their metadata and data, and support RDMA network by substituting their communication modules. For Assise, Assise-1r/Assise-3r indicates that there are one/three hot replicas among cluster nodes. We also compare Hydra with two local PM-aware file systems: EXT4-DAX [40] and NOVA [17], [21].

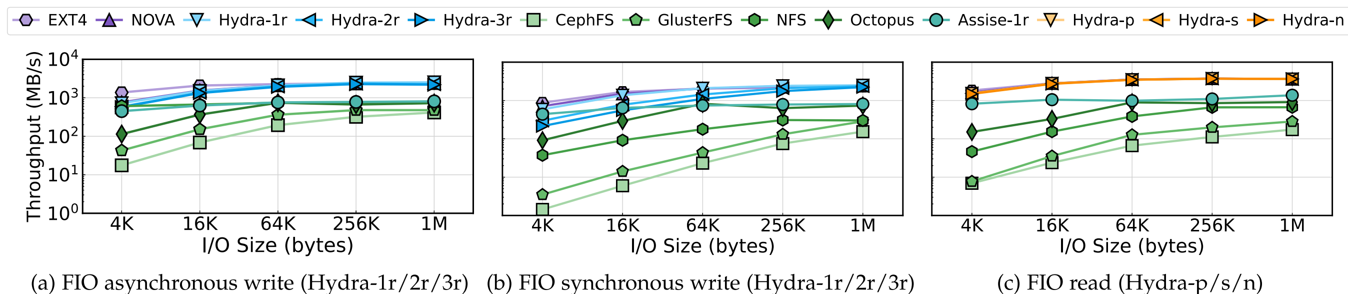
For Hydra, we vary the replication factor to illustrate how performance changes with different storage configurations. *Hydra-1r*/*Hydra-2r*/*Hydra-3r* indicates that there are one primary copy and none/one/two secondary copies among cluster nodes. By default, each workload runs on the local Hydra node (primary node). For read-dominated workloads, the replication factor of Hydra does not impact the read throughput on the primary node. For these workloads, we setup Hydra-2r on two cluster nodes, and then pre-load file data on one of the nodes. During the experiments, we compare the throughput of the workloads executed on the primary node (**Hydra-p**), the secondary node (**Hydra-s**), and a newly connected node that needs to fetch all the working set files from the primary node (**Hydra-n**). We run each workload three times and report the average across these runs.

7.2 Microbenchmarks

We evaluate the read/write throughput of Hydra using FIO [41] benchmark. We perform single-threaded random I/O operations to a 1 GB file with various I/O sizes for one minute, and report the average throughput. In the synchronous write workload, an `fsync` is called after every write operation.

Fig. 6 shows the read/write throughput over various I/O sizes. Hydra leverages the data access locality by fully exploiting the performance advantages of local PM to minimize I/O latency. Therefore, Hydra's throughput is close to NOVA for various I/O sizes in the file read and asynchronous write workloads. For asynchronous writes (Fig. 6a), Hydra-3r achieves 77% of NOVA's throughput with 4 KB I/O size. The gap narrows rapidly with the increasing I/O size. Although each file write initiates an asynchronous RPC request to each secondary node, secondary nodes synchronize the update only through one-sided RDMA reads, which bypasses the

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.



(a) FIO asynchronous write (Hydra-1r/2r/3r) (b) FIO synchronous write (Hydra-1r/2r/3r) (c) FIO read (Hydra-p/s/n)

Fig. 6. *FIO performance (log scale).* Hydra achieves high throughput with increasing I/O sizes, and outperforms CephFS, GlusterFS, NFS, and Octopus by a large margin in all three workloads, especially for synchronous writes.

CPU of the primary node. When the I/O size is increased, the asynchronous RPC requests are issued less frequently. Hence, Hydra maintains high throughput for large file writes.

Synchronous writes (Fig. 6b), however, stall each write until all replications are completed. When the I/O size is 4 KB, Hydra-2r and Hydra-3r achieve 42% and 30% of NOVA's throughput, respectively. Frequent synchronization introduces heavy network traffic for data replication. However, Hydra utilizes differential file update scheme to only synchronize the latest logs with efficient RDMA reads. Besides, when the I/O size is increased, the major bottleneck shifts from log replication to data replication, Hydra is able to batch multiple RDMA requests to reduce data synchronization overhead.

For file reads (Fig. 6c), the read throughputs of all Hydra configurations are very close. The secondary node possesses another complete copy of files, thus its throughput is close to the primary node's. For the newly connected node (Hydra-n), the file does not locally exist initially. When a data segment is read for the first time, Hydra will replicate that data segment to the local node so that future reads become local. At the same time, Hydra will also initiate an asynchronous request to replicate the whole file to the local PM.

Octopus performs 5.1× and 3.1× worse than Hydra-3r for reads and writes on average, respectively. Although Octopus bypasses the kernel buffer cache, file I/O still incurs significant overhead due to the FUSE indirection layer. Besides, Octopus accesses file data only through RDMA networks without client-side caching, which leads to higher latency. Since `fsync` in Octopus is a no-op, its synchronous and asynchronous write throughputs are close.

Hydra outperforms CephFS, GlusterFS, and NFS by a large margin. These three distributed file systems introduce substantial software overhead to the critical I/O path due to their disk-based design, which makes them unable to leverage the direct access feature of PM and the full potential of RDMA networks. The high remote synchronization overhead further lowers their synchronous write throughput. On average, the overall FIO performance of Hydra-3r outperforms CephFS, GlusterFS, and NFS by 54.4×, 32.9×, and 7.3×, respectively.

7.3 Macrobenchmarks

We evaluate the multi-threaded performance and scalability of Hydra with three Filebench [42] workloads: fileserver, webproxy, and varmail. Table 1 summarizes the characteristics of these workloads. The dataset size of each workload is set to 32 GB for a fair comparison.

Fig. 7 shows the Filebench throughput of Hydra on one of the cluster nodes along with that of other file systems. We observe that Hydra demonstrates good scalability in all three workloads, and achieves the best performance among all distributed file systems. The performance gap between Hydra-1r and NOVA is within 5%. However, file replication has different impacts on Hydra-2r and Hydra-3r among the three workloads. On average, Hydra-3r achieves 91%, 88%, and 78% of NOVA's performance in fileserver, webproxy, and varmail, respectively. The throughput of webproxy and varmail saturated with eight threads due to the limited write scalability of Optane DCPMM.

Fileserver emulates the I/O activity of a simple file server. Hydra performs close to NOVA, and its throughput highly scales along with the number of threads. Hydra utilizes the differential file update scheme to lower the CPU overhead when dealing with the small asynchronous writes in fileserver. Therefore, Hydra achieves high throughput in all configurations. Hydra-3r outperforms CephFS, GlusterFS, NFS, and Assise by 47.3×, 21.6×, 5.1×, and 1.7× on average, respectively.

Webproxy is a read-intensive workload, which involves creates, deletes, appends, and repeated reads to files. Different from the fileserver and varmail workloads, the webproxy workload has a large directory width, which makes the efficiency of directory updates critical under such a scenario. Fortunately, Hydra only needs to synchronize the newly created dentry logs thanks to the differential file update scheme. Moreover, these dentry logs are transferred via outbound RDMA reads, which mitigates the transmission latency and improves scalability. For webproxy, Hydra achieves the highest throughput among all distributed file systems.

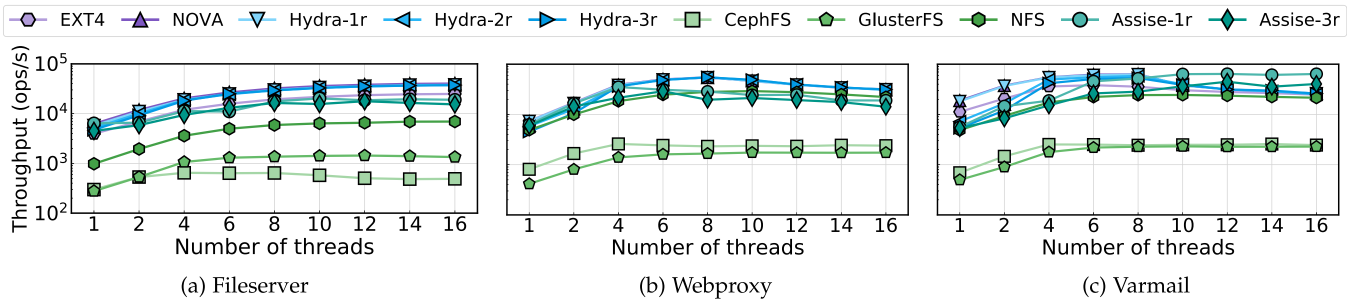
Varmail emulates an email server with frequent synchronous writes. During each `fsync`, Hydra posts synchronous RPC requests to secondary nodes concurrently and waits for their completion. Therefore, as shown in Fig. 7c, the throughputs of Hydra-2r and Hydra-3r are close. RDMA request batching and RPC classification also lower the overhead of

TABLE 1
Filebench Workload Characteristics

Workload	Average file size	# of files	I/O size (R/W)	Directory width	R/W ratio
Fileserver	2MB	16K	16KB/16KB	20	1:2
Webproxy	2MB	16K	1MB/16KB	1M	5:1
Varmail	2MB	16K	1MB/16KB	1M	10:1

The workloads have different read/write ratios and access patterns.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.



(a) Fileserver (b) Webproxy (c) Varmail

Fig. 7. *Filebench performance (log scale).* Hydra demonstrates good multi-core scalability in all three Filebench workloads, and achieves the best overall performance among all distributed file systems. The throughput lines of Hydra-1r and NOVA are highly overlapped.

file synchronization. CephFS and GlusterFS introduce high synchronization overhead during `fsync` due to their disk-based design. Overall, Hydra outperforms CephFS, GlusterFS, and NFS by 13.9×, 16.5×, and 1.6× on average, respectively.

Assise-1r achieves the best performance among all distributed file systems in the varmail workload. Synchronous updates to the log records on Assise-1r are mostly superseded by subsequent updates, which drastically reduces the synchronization overhead of Assise. The hierarchical lease design of Assise also alleviates the network bottlenecks through localized lease management. However, Assise exhibits higher replication overhead than Hydra due to its RDMA write based chain-replication design. On average, Hydra-3r outperforms Assise-3r by 1.6× in all three Filebench workloads.

Storage Overhead. We evaluate the storage overhead of Hydra by gathering the PM usage of each node with respect to different replication factors. We measure the PM usage of file replication after executing the fileserver workload on one of the Hydra nodes. Since all Hydra nodes in our cluster have the same PM capacity, Hydra distributes file replications (secondary copies) evenly among other cluster nodes. For Hydra-2r/3r, we observe that the storage overhead of replication is slightly higher than two/three times the size of the dataset on the primary node (2.08×/3.17×). The additional usage overhead comes from a small number of auxiliary directories and file inodes on the secondary nodes. They are used to maintain the directory tree structure on each Hydra node. This allows Hydra to continue file service of its local files even if all other Hydra nodes are lost.

Load Balancing. As we discussed above, the load of Hydra is properly balanced for static cluster configurations. To explore whether Hydra rebalances after node addition, we run Filebench workloads with 16 threads on Hydra-3r with three nodes, and then add another node into the cluster and rerun the workload. We compare the performance and PM usage of two load-balancing techniques: *lazy balancing* (Hydra-L) and *eager balancing* (Hydra-E). Lazy balancing is the default strategy of Hydra that steers the new file writes to the new node, but it does not migrate existing files from old nodes unless their PM usages are high. Eager balancing, on the other hand, is employed by some distributed file systems (such as the CRUSH algorithm [43] of CephFS) that initiate background migration when nodes are added to the cluster. The throughputs of Hydra-L are close to previous runs (within 1%), whereas Hydra-E's throughputs decline by 4% on average due to minor performance impact from node communication and background migration. However, Hydra-E reaches a balanced data distribution sooner than Hydra-L, which will benefit file accesses from other nodes in the near future, especially when Hydra is deployed in a large-scale cluster. We leave the optimizations for a fast and low-overhead load balancing approach as future work.

7.4 RocksDB

We illustrate the high efficiency of metadata and data replication with RocksDB [44], an embedded key-value store based on LSM-tree. We measure the performance of RocksDB with three workloads from db_bench: sequential read (*readseq*), random read (*readrandom*), and random update (*updaterandom*). For each workload, we first load the database (*fillseq*) with 10M key-value entries on the local node, and then report the throughput of running 10M key-value operations on the local node as well as the remote node with the same dataset. For each workload, the key size is set to 16 bytes, and the value size is set to 4 KB.

Fig. 8 shows the RocksDB throughput. Hydra has the highest throughput among all distributed file systems. For local access, the performance gap between Hydra and NOVA is within 4%. Less frequent file I/O and differential file update scheme help Hydra to hide the replication overhead from foreground file operations. In the sequential read workload, Hydra-n performs 41%, 8.3×, 27% better than CephFS, GlusterFS, and NFS, respectively. During file reads, CephFS and NFS use the kernel buffer cache to achieve high throughput. GlusterFS performs worst due to its high data management overhead. Compared with Hydra-1r, the performance of Hydra-n dropped by 24% for remote random reads. This is because Hydra replicates file data in the background according to the order of file logs, which are sequentially written during the loading period of the workload. Consequently, random reads incur more access misses than local reads.

For local random updates, Hydra takes the advantage of local PM to achieve high performance. CephFS performs worst for its long OSD-based I/O path and mis-prefetching due to the random access pattern. When the random updates are handled by the remote node, all the distributed file systems except Hydra have to go through multiple cache layers to synchronize file data. Not only does Hydra provide direct PM access to file data, RDMA request batching and RPC classification also enable Hydra to perform efficient file transmission by fully utilizing the performance benefits of RDMA networks. For remote random updates, Hydra-n outperforms CephFS, GlusterFS, and NFS by 4.9×, 2.1×, and 2.7×, respectively.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

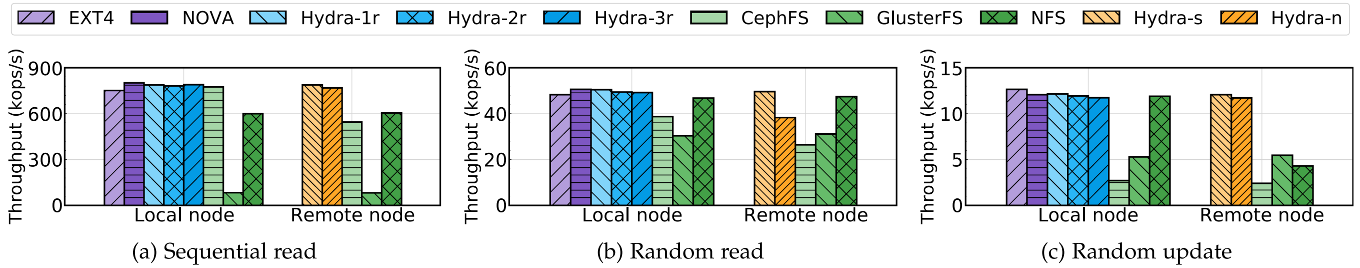


Fig. 8. *RocksDB performance.* The performance gap between local and remote file access of Hydra is small due to the differential file update scheme and RDMA optimization techniques adopted by Hydra.

7.5 MongoDB

We further analyze the parallel performance of Hydra by measuring MongoDB [45], a popular NoSQL database. In MongoDB, each update is logged to a journal file. After the log is persisted, MongoDB writes the update to a memory-mapped database file, and it calls `fsync` to flush its memory to persistent storage periodically. We run all six YCSB [46] workloads on MongoDB. In YCSB, the size of each key-value pair is 1KB (by default). For each workload, we use a total of 1M key-value entries and 10M operations. The number of threads is set to 8.

Fig. 9 shows the MongoDB throughput on five comparison file systems, Hydra with one and three replication configurations, as well as the aggregated throughput when simultaneously running the workloads in private directories on all four cluster nodes of Hydra. For single-node performance, Hydra-1r and Hydra-3r perform close to NOVA, and is 18%, 32%, and 4% faster than CephFS, GlusterFS, and NFS on average, respectively. CephFS and GlusterFS suffer from complicated data management layer, while NFS utilizes the kernel buffer cache to reduce access latency. Due to the less intensive I/O activities, Hydra outperforms these conventional distributed file systems by a smaller margin.

The aggregated throughput of Hydra highly scales with the number of nodes, indicating that decentralization takes full advantage of local PM on each node to achieve high parallel performance. On average, the aggregated throughput of Hydra-3r on four nodes yields 3.8× NOVA's throughput. For write-intensive workloads (workload A and F), Hydra takes the advantage of differential file update scheme to replicate differential file logs and data to the secondary nodes, which effectively mitigates the heavy journal synchronization overhead. Moreover, the RDMA optimization techniques adopted by Hydra further lower the CPU overhead of the decentralized cluster nodes.

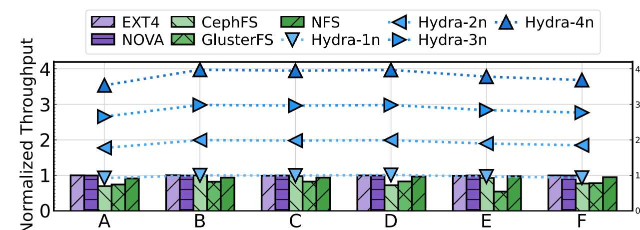


Fig. 9. *MongoDB performance under YCSB workloads.* The results are normalized to NOVA's throughput. Hydra's parallel performance is highly scalable with the number of nodes.

7.6 MDTest

We evaluate the metadata performance of the file systems with the metadata benchmark MDTest [47]. We measure the throughput of the `creation`, `stat`, and `deletion` operations to directories and files. We configure the MDTest workload to operate in a directory tree and process 0.1 million files.

Fig. 10 depicts the throughput of directory and file metadata operations. Hydra achieves the highest overall metadata throughput among all distributed file systems. For directory and file creation, the performance of Hydra-1r is close to NOVA. As for Hydra-2r and 3r, their throughputs drop by 37% on average due to RPC request transmission and processing. Meanwhile, all the disk-based file systems have to transmit the metadata requests to the server nodes through a complex software path designed for ethernet and disks, leading to orders of magnitude higher latencies than Hydra. After dir/file creation, Hydra caches them on the local node to accelerate future accesses. As a result, the throughputs of reading directory and file status of Hydra are similar to local PM-based file systems. CephFS and NFS utilize the kernel buffer cache to provide higher metadata read performance than GlusterFS. For directory and file deletion, Hydra sends synchronous RPC requests to commit the deletion and asynchronous requests to free up storage space (described in Section 4.4). Hence, Hydra has a much smaller performance overhead. Meanwhile, CephFS and GlusterFS perform worse than NFS due to their strict persistence requirement and inefficient software design, respectively.

7.7 Recovery Overhead

We evaluate the recovery overhead of Hydra by measuring its recovery time and the time it takes for Hydra to restore full I/O performance. To simulate a crash, we first load a file set of eight 4 GB files using FIO, and then unmount the file system. After that, we measure the time it takes for Hydra and other file systems to recover their runtime state. Finally, we monitor Hydra's I/O throughput over time until it fully recovers its performance.

As shown in Table 2, the recovery time of Hydra, as well as that of other file systems, is at sub-second level. For Hydra's recovery, it needs to read the superblock to recover the file system metadata, such as the utilization of each PM allocator and inode table, and it also establishes RDMA connections to all the other cluster nodes. As a result, the recovery time of Hydra is slightly longer than that of NOVA. Since we adopt a lazy file synchronization approach that only updates the local stale directories and files upon access, the recovery time does not scale with increasing data sizes.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

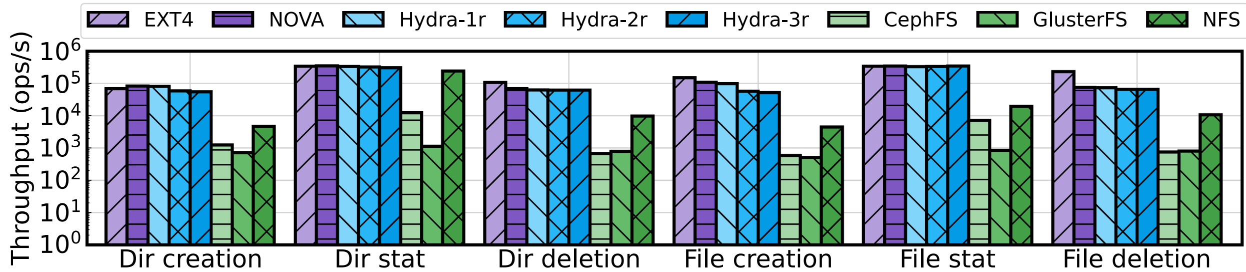


Fig. 10. *Throughput of file system metadata operations.* Hydra provides orders of magnitude higher throughput than disk-based distributed file systems.

To investigate the impact of lazy file synchronization on the runtime performance of Hydra, we measure the dynamic throughput during the first 8 seconds of FIO read with 8 threads on a newly-connected Hydra node (Hydra-n) after fail-over. The FIO workload is configured to issue 50% random read requests and 50% random write requests to the files. Since Hydra-n does not have any local copies of the files in the file set, it must obtain the file data from other nodes to serve local I/O requests, which temporarily impacts the performance. As shown in Fig. 11, the I/O throughput is affected by data replication during the first 2-4 seconds after fail-over. With the increasing I/O size, the replication overhead is decreased due to less metadata RPC transmission, reducing the time it takes for Hydra to fully recover its I/O performance.

8 Related Work

The emergence of persistent memory and high-speed RDMA networks provides an opportunity of having large high-performance distributed storage space across servers. In this section, we briefly describe the works that are closely related to Hydra.

Persistent Memory File Systems. The promising features of emerging persistent memory have stimulated the design and implementation of several PM-based file systems ([12], [13], [14], [17], [19], [20], [21], [48], [49], [50]). Among them, NOVA [17] is a log-structured kernel-space file system that stores metadata and data in PM and maintains file indexes in DRAM. NOVA combines logging, light-weight journaling, and copy-on-write techniques to provide strong atomicity guarantees to both metadata and data. Strata [49], on the other hand, is a user-space tiered file system that manages data accesses in user space and processes metadata in kernel space. Strata utilizes the byte-addressability of PM to store file logs efficiently and asynchronously digest the logs to storage devices. Compared to these file systems, SplitFS [19] utilizes a user-space library file system and a kernel PM file system to handle data and metadata operations, respectively. It uses EXT4-DAX for metadata management and introduces a new relink primitive to speed up file appends and atomic data operations. Linux has also added support for persistent memory to existing file systems, such as EXT4-DAX [40] and XFS-DAX [51] to allow direct access to persistent memory, bypassing DRAM page cache to improve performance.

Distributed File Systems. Existing disk-based distributed file systems, such as CephFS [4], HDFS [5], GlusterFS [6], Lustre [52], and GFS [53], are employed in large scale datacenter deployments. Several decentralized file systems, such as GPFS [54], Farsite [55] and DeltaFS [56], have also been proposed to achieve high scalability without dedicated metadata servers. These file systems focus on providing high availability and scalability by distributing and replicating data blocks in large chunks across servers. In order to adapt to RDMA networks, these disk-based file systems simply replace the communication module with an RDMA library.

Octopus [2] is a user-space distributed file system that uses FUSE [57] for file I/O. Octopus introduces self-identified RPC and collect-dispatch transaction to provide low-latency metadata and data access. However, Octopus uses a static hash function for file placement, which limits its scalability and prevents it from running complex workloads. Besides, Octopus does not provide either metadata or data replication, which makes it vulnerable to server failure.

Orion [3] is a kernel-space distributed file system that leverages RDMA to provide low latency file access. For file I/O, Orion uses local DRAM read cache and PM write buffer to reduce network overhead. As for metadata, Orion replicates file metadata with a Mojim-like [58] technique. All the metadata updates flow to a central metadata server, and then propagated to a mirror server. Since the metadata server handles all metadata updates, the scalability of metadata access becomes a bottleneck for Orion.

Assise [11] is built on top of a cache coherence layer called CC-NVM, which provides linearizability and crash consistency. Similar to Hydra, Assise utilizes persistent caching in client-local PM to achieve fast fail-over and maximizes locality. However, Assise coordinates leases and fault tolerate service via a centralized cluster manager. Besides,

TABLE 2
File System Recovery Time

File System	NOVA	Hydra-1r	Hydra-2r	Hydra-3r
Recovery Time	225 ms	241 ms	239 ms	246 ms
File System	EXT4	CephFS	GlusterFS	NFS
Recovery Time	122 ms	427 ms	386 ms	183 ms

Hydra's recovery time remains constant because it needs a fixed amount of work upon recovering.

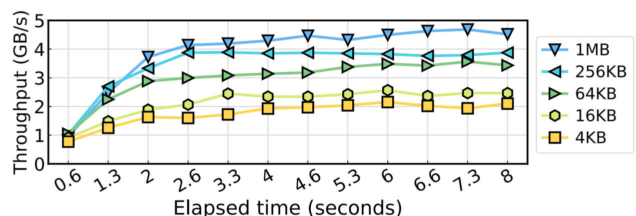


Fig. 11. *I/O throughput of Hydra after fail-over with different I/O sizes.* Once a file has been completely replicated to the local Hydra node, its I/O throughput becomes stable.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

during node recovery, Assise invalidates every block from every file that has been written since its crash, which significantly increases the recovery overhead compared with the differential file update scheme proposed by Hydra.

To the best of our knowledge, Octopus, Orion, and Assise are the only existing distributed file systems that are designed for PM and RDMA networks. Nevertheless, all three file systems adopt centralized cluster management and/or data management architecture that limits scalability. As for file replication, both Orion and Assise replicate files via RDMA writes from the source node, while Hydra performs replication via RDMA reads from the target node with lower overhead. Octopus does not provide either metadata or data replication.

Distributed Persistent Memory Systems. Many existing systems explore how to use RDMA to accelerate shared memory access to applications. Hotpot [26] provides a global, shared persistent memory space to applications and uses a multistage commit protocol to ensure data durability and reliability. AsymNVM [22] is a framework based on an asymmetric persistent memory architecture. It uses operation logs to reduce the stall time due to remote persistency, and enables efficient batching and caching. Flatstore [29] is a PM-based log-structured storage engine that decouples key-value store into a volatile index and a log-structured storage to amortize the persistence overhead. FileMR [59] is a new memory region abstraction that combines RDMA memory regions and files, merging the persistent memory file system and RDMA control plane. Clover [60] is a key-value store system that separates the distributed data plane and the centralized metadata plane, exploiting the benefits of disaggregating persistent memory.

Several RPC approaches have been proposed to optimize performance for RDMA networks as well. DaRPC [61] distributes computation, network, and RPC resources across CPU cores and memory to achieve high aggregated throughput. FaSST [24] is an RDMA-based RPC system built entirely with two-sided RDMA verbs on unreliable datagram (UD) transport to reduce the number of QPs. ScaleRPC [28] proposes connection grouping to reduce NIC cache contention for outbound messages and implements virtualized mapping to improve CPU cache efficiency for inbound messages.

9 Conclusion

We have implemented and described Hydra, a decentralized file system for high-speed PM and RDMA networks. By exploiting the performance advantages of local PM, Hydra leverages data access locality to achieve high performance. To accelerate file transmission, file metadata and data are decoupled and updated differentially through one-sided RDMA reads. We further minimize network overhead by introducing RDMA request batching and RPC classification mechanisms. The decentralized design enables Hydra to tolerate node failures and achieve load balancing. Our evaluation shows that Hydra significantly outperforms existing distributed file systems, and shows good scalability on multi-threaded and parallel workloads.

Acknowledgment

The author would like to thank the anonymous reviewers for their valuable feedback and insightful suggestions.

References

[1] Intel, "Intel optane DC persistent memory," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
[2] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: An RDMA-enabled distributed persistent memory file system," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 773–785.
[3] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A distributed file system for non-volatile main memory and RDMA-capable networks," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 221–234.
[4] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Oper. Syst. Des. Implementation*, 2006, pp. 307–320.
[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[6] A. Davies and A. Orsaria, "Scale out with glusterFS," *Linux J.*, vol. 2013, no. 235, 2013, Art. no. 1.
[7] A. Kalia, D. Andersen, and M. Kaminsky, "Challenges and solutions for fast remote persistent memory access," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 105–119.
[8] J. Izraelevitz *et al.*, "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv: 1903.05714*.
[9] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 169–182.
[10] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 496–508.
[11] T. E. Anderson *et al.*, "Assise: Performance and availability via client-local NVM in a distributed file system," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 1011–1027.
[12] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ.*, 2009, pp. 133–146.
[13] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 14.
[14] S. R. Dulloor *et al.*, "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–15.
[15] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 12.
[16] E. H.-M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "A new design of in-memory file system based on file virtual address framework," *IEEE Trans. Comput.*, vol. 65, no. 10, pp. 2959–2972, Oct. 2016.
[17] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 323–338.
[18] M. Dong and H. Chen, "Soft updates made simple and fast on non-volatile memory," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 719–731.
[19] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing software overhead in file systems for persistent memory," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 494–508.
[20] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable persistent memory file system with kernel-userspace collaboration," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 81–95.
[21] J. Xu *et al.*, "NOVA-fortis: A fault-tolerant non-volatile main memory file system," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 478–496.
[22] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 757–773.
[23] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be general and fast," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 1–16.
[24] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs," in *Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation*, 2016, pp. 185–201.
[25] S.-Y. Tsai and Y. Zhang, "Lite kernel RDMA support for datacenter applications," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 306–324.
[26] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *Proc. Symp. Cloud Comput.*, 2017, pp. 323–337.

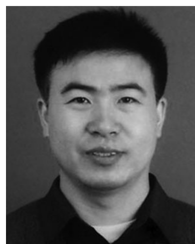
Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.

[27] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed os for hardware resource disaggregation," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 69–87.
[28] Y. Chen, Y. Lu, and J. Shu, "Scalable RDMA RPC on reliable connection with efficient resource sharing," in *Proc. 14th Eur. Syst. Conf.*, 2019, pp. 1–14.
[29] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 1077–1091.
[30] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 295–306.
[31] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "RFP: When RPC is faster than server-bypass with RDMA," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 1–15.
[32] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.
[33] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.
[34] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, Oct. 2001.
[35] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, 1997, pp. 654–663.
[36] Intel, "Intel data direct I/O technology," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html
[37] S. Li *et al.*, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 476–488.
[38] X. Hu, M. Ogleari, J. Zhao, S. Li, A. Basak, and Y. Xie, "Persistence parallelism optimization: A holistic approach from memory bus to RDMA network," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 494–506.
[39] T. Haynes and D. Noveck, "Network file system (NFS) version 4 protocol," 2015. [Online]. Available: https://tools.ietf.org/html/rfc7530
[40] M. Wilcox, "Add support for NV-Dimms to ext4," 2014. [Online]. Available: https://lwn.net/Articles/613384/
[41] J. Axboe, "Flexible I/O tester," 2016. [Online]. Available: https://github.com/axboe/fio
[42] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *USENIX; Login*, vol. 41, no. 1, pp. 6–12, 2016.
[43] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proc. ACM/IEEE Conf. SuperComput.*, 2006, pp. 31–31.
[44] Facebook, "Rocksdb," 2018. [Online]. Available: http://rocksdb.org
[45] MongoDB, "Mongodb," 2018. [Online]. Available: http://www.mongodb.org/
[46] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[47] Mdtest, "Mdtest HPC benchmark," 2015. [Online]. Available: https://sourceforge.net/projects/mdtest/
[48] X. Wu and A. Reddy, "Scmfs: A file system for storage class memory," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 39.
[49] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 460–477.
[50] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 478–493.
[51] D. Chinner, "XFS: DAX support," 2015. [Online]. Available: https://lwn.net/Articles/635514/
[52] Lustre, "Lustre file system," 2017. [Online]. Available: http://www.lustre.org/
[53] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Oper. Syst. Princ.*, 2003, pp. 29–43.
[54] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, Art. no. 19.
[55] J. J. Douceur and J. Howell, "Distributed directory service in the farsite file system," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006.
[56] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale file systems scale better without dedicated servers," in *Proc. 10th Parallel Data Storage Workshop*, 2015, pp. 1–6.
[57] libfuse, "FUSE (filesystem in userspace)," 2021. [Online]. Available: https://github.com/libfuse/libfuse/
[58] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2015, pp. 3–18.
[59] J. Yang, J. Izraelevitz, and S. Swanson, "FileMR: Rethinking RDMA networking for scalable persistent memory," in *Proc. 17th USENIX Symp. Netw.ed Syst. Des. Implementation*, 2020, pp. 111–125.
[60] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 33–48.
[61] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "DaRPC: Data center RPC," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.

Shengan Zheng received the BS and PhD degrees from Shanghai Jiao Tong University, in 2014 and 2019, respectively. He is currently an assistant professor with Shanghai Jiao Tong University. His research interests include persistent memory-based storage systems, file systems, and distributed systems.

Jingyu Wang received the BS degree from the University of Electronic Science and Technology of China, in 2018. She is currently working toward the PhD degree in computer science with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. Her research interests include the area of non-volatile memories and distributed file systems.

Dongliang Xue received the PhD degree in computer science from Shanghai Jiao Tong University, in 2019. He is currently an assistant research fellow in computer science with the Department of Computer Science and Engineering, Shanghai Jiao Tong University since July in 2019. His research interests include in-memory computing, cloud computing, and non-volatile memory.



Jiwu Shu (Fellow, IEEE) received the PhD degree in computer science from Nanjing University, in 1998, and finished his postdoctoral position research with Tsinghua University, in 2000. Since then, he has been teaching with Tsinghua University, and is currently a professor with the Department of Computer Science and Technology, Tsinghua University. His current research interests include network storage systems, non-volatile memory-based storage systems, storage security and reliability, and parallel and distributed computing.

Linpeng Huang (Senior Member, IEEE) received the MS and PhD degrees in computer science from Shanghai Jiao Tong University, in 1989 and 1992, respectively. He is currently a professor in computer science with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include distributed systems and service-oriented computing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 30,2022 at 07:54:06 UTC from IEEE Xplore. Restrictions apply.