Octopus+: An RDMA-Enabled Distributed Persistent Memory File System

BOHONG ZHU, YOUMIN CHEN, QING WANG, YOUYOU LU, and JIWU SHU,
Tsinghua University

Non-volatile memory and remote direct memory access (RDMA) provide extremely high performance in storage and network hardware. However, existing distributed file systems strictly isolate file system and network layers, and the heavy layered software designs leave high-speed hardware under-exploited.

In this article, we propose an RDMA-enabled distributed persistent memory file system, Octopus+, to redesign file system internal mechanisms by closely coupling non-volatile memory and RDMA features. For data operations, Octopus+ directly accesses a shared persistent memory pool to reduce memory copying overhead, and actively fetches and pushes data all in clients to rebalance the load between the server and network. For metadata operations, Octopus+ introduces self-identified remote procedure calls for immediate notification between file systems and networking, and an efficient distributed transaction mechanism for consistency. Octopus+ is enabled with replication feature to provide better availability. Evaluations on Intel Optane DC Persistent Memory Modules show that Octopus+ achieves nearly the raw bandwidth for large I/Os and orders of magnitude better performance than existing distributed file systems.

CCS Concepts: • **Computer systems organization → Client-server architectures**; • **Software and its engineering → File systems management**; **Distributed memory;**

Additional Key Words and Phrases: Storage system, distributed system, remote direct memory access, persistent memory

ACM Reference format:
Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. 2021. Octopus+: An RDMA-Enabled Distributed Persistent Memory File System. *ACM Trans. Storage* 17, 3, Article 19 (August 2021), 25 pages.
https://doi.org/10.1145/3448418

1 INTRODUCTION

The in-memory storage and computing paradigm emerges as both HPC and big data communities are demanding extremely high performance in data storage and processing. Recent in-memory storage systems, including both database systems (e.g., SAP HANA [6]) and file systems (e.g., Alluxio [33]), have been used to achieve high data processing performance. With the emerging

This work is supported by National Key Research & Development Program of China (grant 2018YFB1003301) and the National Natural Science Foundation of China (grant 61832011, 61772300).
Authors' addresses: B. Zhu, Y. Chen, Q. Wang, Y. Lu, and J. Shu (corresponding author), Tsinghua University; emails: {zhubh18, chenym16, q-wang18}@mails.tsinghua.edu.cn, {luyouyou, shujw}@tsinghua.edu.cn.

1553-3077/2021/08-ART19 $15.00
https://doi.org/10.1145/3448418

non-volatile memory (NVM) technologies, such as phase change memory [30, 48, 65], resistive RAM, and Intel's Optane DC Persistent Memory [19], data can be stored persistently in main memory level (i.e., *persistent memory*). New local file systems, including PMFS [15], NOVA [58], SplitFS [24], and Strata [29], were built recently to exploit the byte-addressability or persistence advantages of NVMs. Their promising results have shown good potential of NVMs in high performance of both data storage and processing.

Meanwhile, **remote direct memory access (RDMA)** technology brings extremely low latency and high bandwidth to the networking. We have measured an average latency and bandwidth of 0.94 us and 12.3 GB/s with a 100-Gbps InfiniBand switch, compared to 75 us and 188 MB/s with **Gigabit Ethernet (GigaE)**. RDMA has greatly improved data center communications or **remote procedure calls (RPCs)** in recent studies [14, 27, 28, 51].

Distributed file systems (DFSs) are trying to support RDMA networks for better performance, but mostly only by substituting the communication module with an RDMA-enabling library. CephFS supports RDMA by using Accelio [1], an RDMA-based asynchronous RPC middleware. GlusterFS implements its own RDMA library for data communication [16]. NVFS [21] is a HDFS variant that is optimized for NVM and RDMA. In addition, Crail [7], a recent DFS from IBM, is built on an RDMA-optimized RPC library, DaRPC [51]. However, these file systems strictly isolate file system and network layers, by only replacing their data management and communication modules without refactoring the internal file system mechanisms. This layered and heavy software design prevents file systems from exploiting the hardware benefits. As we observed, GlusterFS has its software latency that accounts for nearly 100% on NVM and RDMA, whereas it is only 2% on disk. Similarly, it achieves only 24% of raw NVM bandwidth and 11% of raw InfiniBand bandwidth, compared to 76% of the raw disk bandwidth and 74% of the GigaE bandwidth. In conclusion, the *strict isolation* between the file system and network layers makes DFSs too heavy and incompetent to exploit the benefits of emerging high-speed hardware.

In this article, we revisit both data and metadata mechanism designs of the DFS by taking NVM and RDMA features into consideration. We propose an efficient distributed persistent memory file system, Octopus$^+$ (it is called Octopus because the file system performs RDMA just like an octopus uses its eight legs), to effectively exploit the benefits of high-speed hardware. Octopus$^+$ avoids the strict isolation of file system and network layers, and redesigns the file system internal mechanisms by closely coupling with NVM and RDMA features. For the data management, Octopus$^+$ directly accesses a shared persistent memory pool by exporting NVM to a global space. This allows Octopus$^+$ to avoid stacking a DFS layer on local file systems and eliminates redundant memory copies. It also rebalances the server and network loads, and revises the data I/O flows to offload loads from servers to clients in a client-active way for higher throughput. For the metadata management, Octopus$^+$ introduces a self-identified RPC that carries the sender's identifier with the RDMA write primitive for low-latency notification. In addition, it proposes a new distributed transaction mechanism by incorporating RDMA write and atomic primitives. Upon such transaction mechanism, Octopus$^+$ incorporates replication mechanism for high data availability. In Octopus$^+$, metadata and data are replicated to multiple physical servers via different protocols, including an operation-log-based replication approach for small-sized metadata and a client-active replication mechanism for file data, which only introduce limited impact on system performance. As such, Octopus$^+$ efficiently incorporates RDMA into file system designs that effectively exploit hardware benefits.

Our major contributions are summarized as follows:

- We propose novel I/O flows based on RDMA for Octopus$^+$, which directly accesses a shared persistent memory pool without stacked file system layers, and actively fetches or pushes data in clients to rebalance server and network loads.

- We redesign metadata mechanisms leveraging RDMA primitives, including self-identified metadata RPC for low-latency notification, a collect-dispatch distributed transaction for low-overhead consistency, and differential data and metadata replication protocols for high availability.
- We implement OCTOPUS+ and evaluate it on servers with Intel DC Persistent Memory Modules and RDMA networks. Experimental results show that OCTOPUS+ effectively explores the raw hardware performance and significantly outperforms existing RDMA-optimized DFSs.

2 BACKGROUND AND MOTIVATION

2.1 NVM and RDMA

NVM. Byte-addressable NVM technologies, including phase change memory [30, 48, 65], resistive RAM, and Memristor [50], have been studied intensively in recent years. These NVMs have access latency close to that of DRAM while providing data persistence as hard disks. Therefore, NVMs are promising candidates for storing data persistently at the main memory level. Recently, Intel released **Optane DC Persistent Memory Modules (DCPMMs)** [19], which is the first commercially available persistent memory product. Currently, new products come in three capacities: 128, 256, and 512 GB. A single DCPMM has a max read/write bandwidth at 6.6 GB/s and 2.3 GB/s, respectively, and a read/write latency at 305 ns and 169 ns, respectively [22].

Remote direct memory access. RDMA enables low-latency network access by directly accessing memory from remote servers. It bypasses the operating system and supports zero-copy networking, and thus achieves high bandwidth and low latency in network accesses. There are two kinds of commands in RDMA for remote memory access:

(1) *Message semantics*, with typical RDMA `send` and `recv` verbs for message passing, are similar to socket programming. Before sending an RDMA `send` request at the client side, an RDMA `recv` needs to be posted at the server side with an attached address indicating where to store the coming message.

(2) *Memory semantics*, with typical RDMA `read` and `write` verbs, use a new data communication model (i.e., *one-sided*) in RDMA. In memory semantics, the memory address in the remote server where the message will be stored is assigned at the sender side. This removes the CPU involvement of remote servers. The memory semantics provide relatively higher bandwidth and lower latency than the message semantics.

In addition, RDMA provides other verbs, including atomic verbs like `compare_and_swap` and `fetch_and_add` that enable atomic memory access of remote servers.

2.2 Software Challenges on Emerging High-Speed Hardware

In a storage system equipped with NVM and RDMA-enabled networks, the hardware can provide extremely higher performance than traditional media like hard disks and GigaE. Comparatively, overheads caused by the software layer, which were negligible before, compared to slow disk and Ethernet, now account for a significant part in the whole system.

Latency. To understand the latency overhead of existing DFSs, we perform synchronous 1-KB write operations on GlusterFS, and collect latencies respectively in the storage, network, and software parts. The latencies are averaged with 10,000 synchronous writes. Figure 1(a) shows the latency breakdown of GlusterFS on disk (denoted as *diskGluster*) and memory (denoted as *memGluster*). To improve efficiency of GlusterFS on memory, we run memGluster on EXT4-DAX [3], which is optimized for NVM by bypassing the page cache and reducing memory copies. In diskGluster, the storage latency consumes the greatest part, nearly 99% of the total latency. In memGluster, the storage latency percentage drops dramatically to nearly zero. In comparison, the file system software latency becomes the dominant part, almost 99%. Similar trends have also

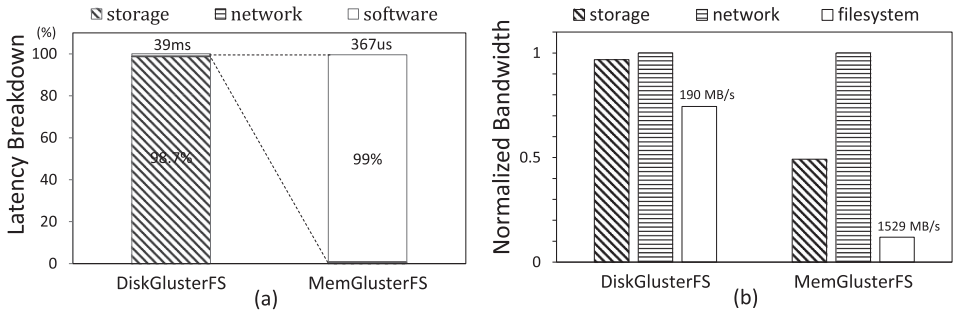


Fig. 1. Software overhead.

been observed in previous studies in local storage systems [52]. Although most DFSs stack the distributed data management layer on another local file system (a.k.a. stacked file system layers), they face more severe software overhead than local storage systems.

Bandwidth. We also measure the maximum bandwidth of GlusterFS to understand the software overhead in terms of bandwidth. In the evaluation, we perform 1-MB write requests to a single GlusterFS server repeatedly to get the average write bandwidth of GlusterFS. Figure 1(b) shows the GlusterFS write bandwidth against the storage and network bandwidths. In diskGluster, GlusterFS achieves a bandwidth that is 76.8% of raw disk bandwidth and 74.4% of raw GigaE bandwidth. In memGluster, GlusterFS's bandwidth is only 23.8% of raw NVM bandwidth and 11% of raw InfiniBand bandwidth. Existing file systems are inefficient in exploiting the high bandwidth of new hardware.

We find that there are four mechanisms that contribute to this inefficiency in existing DFSs. First, data are copied multiple times in multiple places in memory, including user buffer, file system page cache, and network buffer. Although this design is feasible for file systems that are built for slow disks and networks, it has a significant impact on system performance with high-speed hardware. Second, when networking is getting faster, the CPU at server side can easily be the bottleneck when processing requests from a lot of clients. Third, traditional RPC that is based on the event-driven model has relatively high notification latency when hardware provides low-latency communication. Fourth, DFSs have huge consistency overhead in distributed transactions, owing to multiple network round trips and complex processing logic.

As such, we propose to design an *efficient* distributed memory file system for high-speed network and memory hardware, by revisiting the internal mechanisms in both data and metadata management.

3 OCTOPUS$^+$ DESIGN

To effectively explore the benefits of raw hardware performance, Octopus$^+$ closely couples RDMA with file system mechanism designs (Figure 2). Both data and metadata mechanisms are reconsidered:

- *High-throughput data I/O*, to achieve high I/O bandwidth by reducing memory copies with a *shared persistent memory pool*, and improve throughput of small I/Os using *client-active I/Os*.
- *Low-latency metadata access*, to provide a low-latency and scalable metadata RPC with *self-identified RPC*, and decrease consistency overhead using the *collect-dispatch transaction*.

3.1 Overview

Octopus$^+$ is built for a cluster of servers that are equipped with NVM and RDMA-enabled networks. Octopus$^+$ consists of two parts: *clients* and *data servers*.

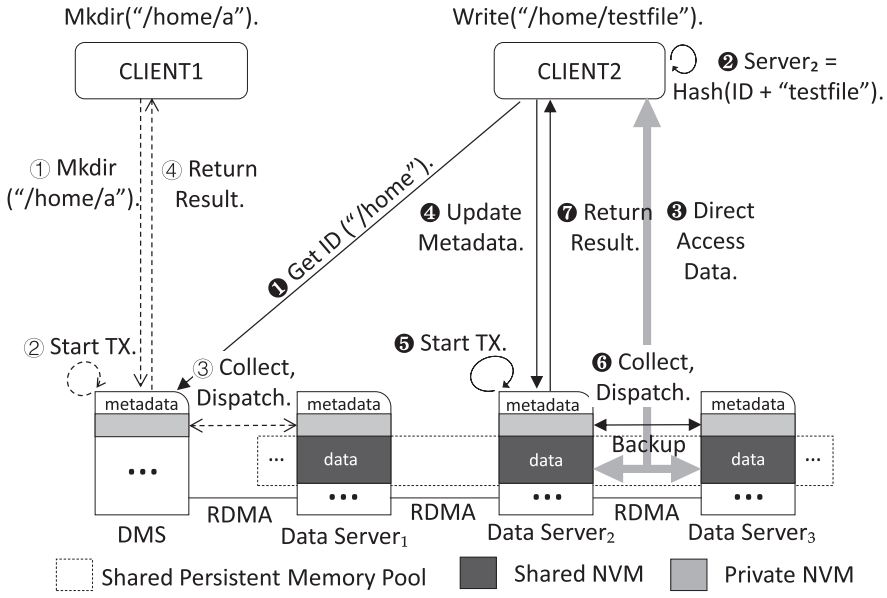


Fig. 2. Octopus+ architecture.

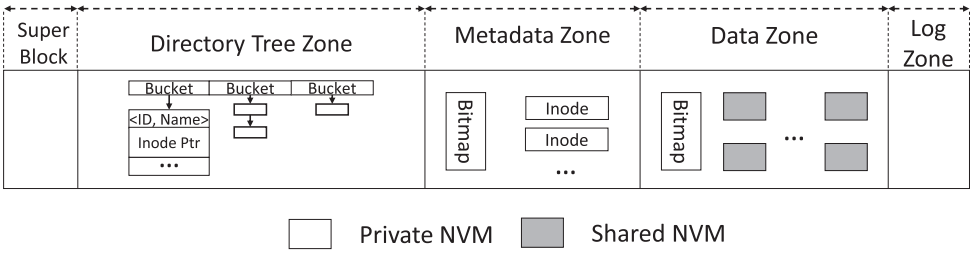


Fig. 3. NVM layout in an Octopus+ node.

At the server side, all directories are kept in one designated directory metadata server (denoted as DMS), and files are distributed to all regular servers in a hash-based fashion (denoted as data server) [34]. The whole NVM area can be briefly divided into data area and metadata area, respectively. The data area is exported and shared among the whole cluster for remote direct data accesses, whereas the metadata area is kept private for consistency reasons. Figure 3 shows the detailed NVM layout of each server, which is organized into five zones: (1) *Super Block* to keep the metadata of the file system, (2) *Directory Tree Zone* to accommodate indexes that assemble the directory tree (it contains a hash table to index files located on this server), (3) *Metadata Zone* to keep the actual file metadata structures like *Inode*, (4) *Data Zone* to keep data blocks, and (5) *Log Zone* to keep transaction log blocks to ensure file system consistency.

A data server keeps metadata and data respectively in the private and shared area; however, Octopus+ accesses these two areas remotely in different ways. For the private metadata accesses, Octopus+ uses optimized RPCs as in existing DFSs. For the shared data accesses, Octopus+ directly reads or writes data objects remotely using RDMA primitives.

With the use of RDMA, Octopus+ removes duplicated memory copies between file system images and memory buffers by introducing the *shared persistent memory pool* (*shared pool* for

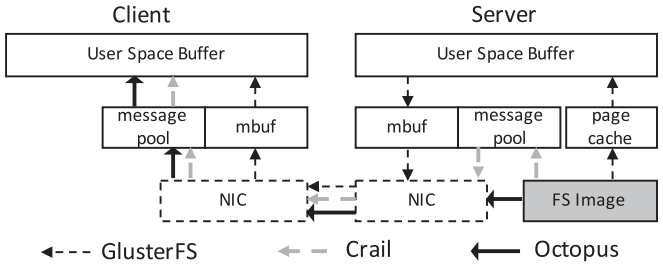


Fig. 4. Data copies in a remote I/O request.

brevity). This shared pool is formed with exported data areas from each data server in the whole cluster (in Section 3.2.1). In current implementation, the memory pool is initialized using a static XML configuration file, which stores the pool size and the cluster information. Octopus$^+$ also redesigns the read/write flows by sacrificing network round trips to amortize server loads using *client-active I/Os* (in Section 3.2.2).

For metadata mechanisms, Octopus$^+$ leverages RDMA write primitives to design a low-latency and scalable RPC for metadata operations (in Section 3.3.1). It also redesigns the distributed transaction to reduce the consistency overhead, by collecting data from remote servers for local logging and then dispatching them to remote sides (in Section 3.3.2). Such transaction mechanism is also adopted in (meta)data replication protocols, which will be illustrated in Section 4.

3.2 High-Throughput Data I/O

Octopus$^+$ introduces a shared persistent memory pool to reduce data copies for higher bandwidth, and actively performs I/Os in clients to rebalance server and network overheads for higher throughput.

3.2.1 Shared Persistent Memory Pool. In a system with extremely fast NVM and RDMA, memory copies account for a large portion of overhead in an I/O request. In existing DFSs, a DFS is commonly layered on top of local file systems. For a read or write request, a data object is duplicated to multiple locations in memory, such as kernel buffer (`mbuf` in TCP/IP stack), user buffer (for storing distributed data objects as local files), kernel page cache (for local file system cache), and file system image in persistent memory (for file storage in a local file system in NVM). As the GlusterFS example shown in Figure 4, a remote I/O request requires the fetched data to be copied seven times including in memory and in the **network interface controller (NIC)** for final access.

Recent local persistent file systems (like PMFS [15] and EXT4-DAX [3]) directly access persistent memory storage without going through the kernel page cache, but it does not solve problems in the DFS cases. With direct access of these persistent memory file systems, only the page cache is bypassed, and a DFS still requires data to be copied six times.

Octopus$^+$ introduces the *shared persistent memory pool* by exporting the data area of the file system image in each server for sharing. The shared pool design not only removes the stacked file system design but also enables direct remote access to file system images without any caching. Octopus$^+$ directly manages data distribution and layout of each server, and does not rely on a local file system. Direct data management without stacking file systems is also taken in Crail [7], a recent RDMA-aware DFS built from scratch. Compared to stacked file system designs like GlusterFS, data copies in Octopus$^+$ and Crail do not need to go through the user space buffer in the server side, as shown in Figure 4.

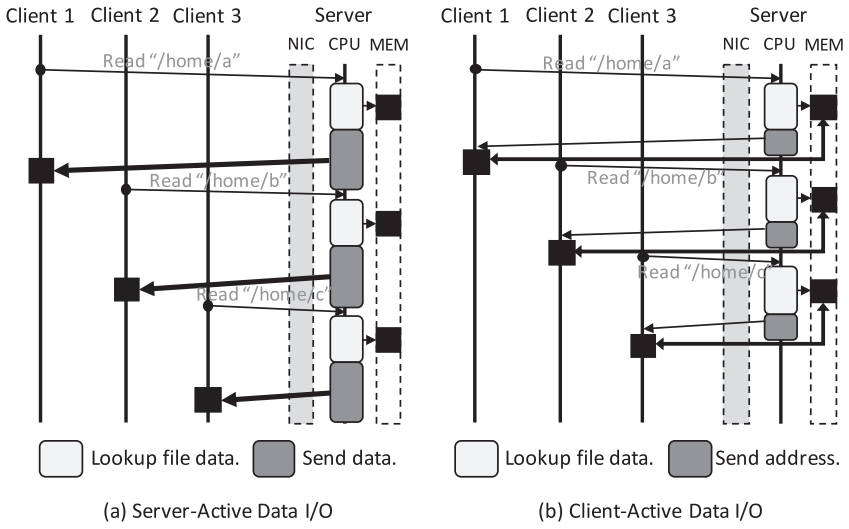


Fig. 5. Comparison of server-active and client-active modes.

Octopus+ also provides a global view of data layout with the shared pool enabled by RDMA. In a data server in Octopus+, the data area in the NVM is registered with *ibv_reg_mr* when the data server joins, which allows the remote direct access to file system images. Hence, Octopus+ removes the use of a `message pool` or a `mbuf` in the server side, which are used for preparing file system data for network transfers. As such, Octopus+ requires data to be copied only four times for a remote I/O request, as shown in Figure 4. By reducing memory copies in NVMs, data I/O performance is significantly improved, especially for large I/Os that incur fewer metadata operations.

3.2.2 Client-Active Data I/O. For data I/O, it is common to complete a request within one network round trip. Figure 5(a) shows a read example. The client issues a read request to the server, and the server prepares data and sends it back to the client. Similarly, a write request can also complete with one round trip. This is called *server-active mode*. Although this mode works well for slow Ethernet, we find that the server is always in high utilization and becomes a bottleneck when new hardware is equipped.

In remote I/Os, the throughput is bounded by the lower one between the network and server throughput. In our cluster, we achieve 5 *million* network IOPS for 1-KB writes but have to spend around 2 us (i.e., 0.5 *million*) for data locating even without data processing. The server processing capacity becomes the bottleneck for small I/Os when RDMA is equipped.

In Octopus+, we propose *client-active mode* to improve server throughput by sacrificing the network performance when performing small size I/Os. As shown in Figure 5(b), in the first step, a client in Octopus+ sends a *read* or *write* request to the server. In the second step, the server sends back the metadata information to the client. Both of the two steps are executed for metadata exchange using the self-identified metadata RPC, which will be discussed next. In the third step, the client reads or writes file data with the returned metadata information, and directly accesses data using RDMA `read` and `write` commands. Since both RDMA `read` and `write` are one-sided operations, meaning that clients can access remote data without participation of CPUs in remote servers, therefore the server in Octopus+ has higher processing capacity. By doing so, a rebalance is made between the server and network overheads. With introduced limited round trips, the load

previously on server side is now offloaded to clients, resulting in higher throughput for concurrent requests.

In addition, Octopus$^+$ uses the per-file read-write lock to serialize the concurrent RDMA-based data accesses. The lock service is based on a combination of CPU and RDMA atomic primitives. To read or write file data, the locking operation is executed by the server locally using CPU atomic instructions. The unlock operation is executed remotely by the client with RDMA atomic verbs after data I/Os. Note that serializability between CPU and RDMA atomic primitives is not guaranteed due to lack of atomicity between the CPU and the NIC [9, 27, 56]. However, in Octopus$^+$, CPU and RDMA atomic instructions are respectively used in the locking and unlocking phases. This isolation prevents the competition between the CPU and the NIC, and thus ensures correctness of parallel accesses.

3.3 Low-Latency Metadata Access

RDMA provides microsecond-level access latencies for remote data access. To explore this benefit in the file system level, Octopus$^+$ refactors the metadata RPC and distributed transaction by incorporating RDMA write and atomic primitives.

3.3.1 Self-Identified Metadata RPC. RPCs are used in Octopus$^+$ for metadata operations. Both message and memory semantic commands can be utilized to implement RPCs:

(1) *Message-based RPC.* In the message-based RPC, a `recv` request is first assigned with a memory address and then initialized in the remote side before the `send` request. Each time an RDMA `send` arrives, an RDMA `recv` is consumed. Message-based RPC has relatively high latency and low throughput. `send`/`recv` in **Unreliable Datagram (UD)** mode provides higher throughput [28] but is not suitable for DFSs due to its unreliable connections.

(2) *Memory-based RPC.* RDMA `read`/`write` have lower latency than `send`/`recv`. Unfortunately, these commands are one-sided, and the remote server is uninvolved. To timely process these requests, the server side needs to scan the message buffers repeatedly to discover new requests. This causes high CPU overhead. Even worse, when the number of clients increased, the server side needs to scan more message buffers, and this in turn increases the processing latency.

To gain benefits of both sides, we propose the self-identified metadata RPC. Self-identified metadata RPC attaches the sender's identifier with the RDMA write request using the RDMA `write_with_imm` command. `write_with_imm` is different from RDMA `write` in two aspects: (1) it is able to carry an immediate field in the message, and (2) it notifies remote side immediately, but RDMA `write` does not. With the first difference, we attach the client's identifier in the immediate data field including both a `node_id` and an `offset` of the client's receive buffer. For the second difference, RDMA `write_with_imm` consumes one receive request from the remote QP (queue pair) and thus gets immediately processed after the request arrives. The identifier attached in the immediate field helps the server to directly locate the new message without scanning the whole buffer. After processing, the server uses RDMA `write` to return data back to the specified address of `offset` in the client of `node_id`. Compared to buffer scanning, this immediate notification dramatically lowers down the CPU overhead when there are a lot of client requests. As such, the self-identified metadata RPC provides low-latency and scalable RPCs that `send`/`recv` and `read`/`write` approaches.

3.3.2 Collect-Dispatch Transaction. A single file system operation, like *mkdir* and *rmdir* in Octopus$^+$, performs updates to multiple servers. In replication mode, such situation will extend to almost all file system operations that require making updates to the file system. Accordingly, distributed transactions are needed to provide concurrency control for simultaneous requests and crash consistency for the atomicity of updates across servers. The **two-phase commit (2PC)**

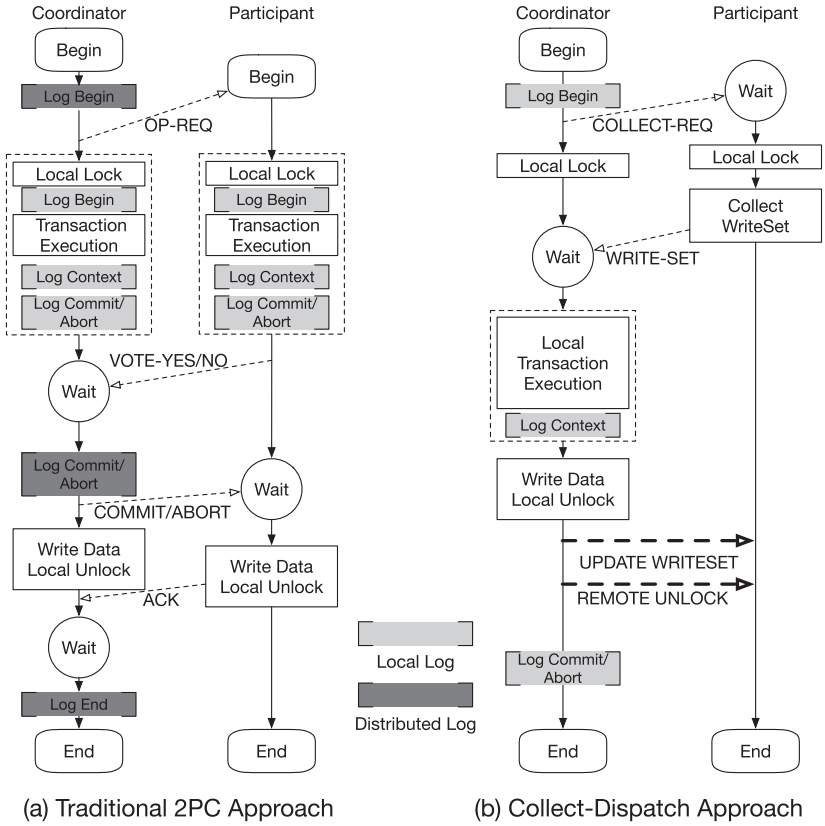


Fig. 6. Distributed transaction.

protocol is usually used to ensure consistency. However, 2PC incurs high overhead due to its distributed logging and coordination for both locks and log persistence. As shown in Figure 6(a), both locking and logging are required in coordinator and participants, and complex network round trips are needed for negotiation for log persistence ordering.

OCTOPUS+ designs a new distributed transaction protocol named *collect-dispatch transaction* leveraging RDMA primitives. The key idea lies in two aspects, respectively in crash consistency and concurrency control. One is *local logging with remote in-place update* for crash consistency. As shown in Figure 6(b), in the collect phase, OCTOPUS+ collects the read and write sets from participants, and performs local transaction execution and local logging in the coordinator. Since participants do not need to keep logging, there is no need for complex negotiation for log persistence between the coordinator and participants, thereby reducing protocol overheads. For the dispatch phase, the coordinator spreads the updated write set to the participants using RDMA `write` and releases the corresponding lock with RDMA atomic primitives, without the involvements of the participants.

The other is a *combination of CPU and RDMA locking* for concurrency control, which is the same as the lock design in the data I/Os in Section 3.2.2. In collect-dispatch transactions, locks are added locally using the `compare_and_swap` instruction in both coordinator and participants. For the unlock operations, the coordinator releases the local lock using the `compare_and_swap` instruction but the remote lock in each participant using the RDMA `compare_and_swap` command.

The RDMA unlock operations do not involve the CPU processing of participants and thus simplify the unlock phase.

As a whole, collect-dispatch requires one RPC, one RDMA write, and one RDMA atomic operation, and 2PC requires two RPCs. Collect-dispatch still has lower overhead, because (1) RPC has higher latency than an RDMA write/atomic primitive, and (2) RDMA write/atomic primitive does not involve CPU processing of the remote side. Thus, we conclude collect-dispatch is efficient, as it not only removes complex negotiations for log persistence ordering across servers but also reduces costly RPC and CPU processing overheads.

4 IMPLEMENTATION

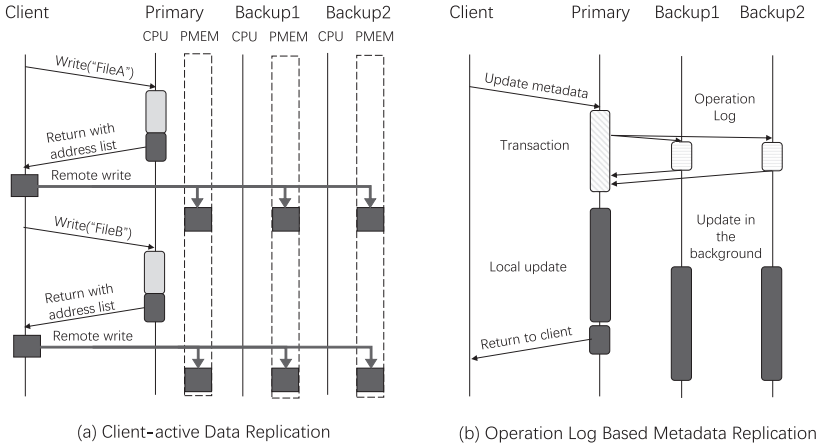
Octopus$^+$ is developed as an RDMA-enabled DFS for persistent memories. For now, Octopus$^+$ is implemented into two modes: user-space library mode and FUSE mode. Applications can access Octopus$^+$ through a user-space library or a POSIX-compliant interface based on FUSE, respectively.[1] To directly operate on NVM devices, Octopus$^+$ configures NVM devices into DAX mode and maps them into the system's address space through system call *mmap*. The rest of this section will demonstrate the implementation details about indexing mechanism, fault tolerance, and file system consistency.

4.1 Indexing Mechanism

Directory tree indexing. Originally in *Octopus* [36], all directories and files are distributed among servers in a hash-based manner. This metadata architecture is inefficient because, for example, to access a file/directory at the (N+1)th level in the directory tree, Octopus has to issue N RPC calls to different servers to access all N parent/ancestor directories, which produces too many network round trips and hinders the metadata performance. Therefore, Octopus$^+$ incorporates a new directory tree indexing mechanism different from its predecessor *Octopus*. As mentioned before, Octopus$^+$ keeps directories in one designated directory metadata server and distributes files among multiple data servers in a hash-based fashion. To serve read and update requests fast and scalable, Octopus$^+$ indexes all directories or files on a server using a chained hash table on NVM. The directory metadata server assigns each directory a unique 64-bit ID when it is created. The ID of the root directory of this file system is set to a reserved number. At the directory metadata server, each item in the hash table represents a directory, which is indexed by a combination of a directory's parent ID and its name. In this item, a *dentry* is recorded. Hence, one can find a directory by recursively resolving the full path starting from the root directory. To locate a file among multiple data servers, Octopus$^+$ first calculates the data server ID from a combination of the parent ID and a file name using a hash function. Then, on that data server, Octopus$^+$ can find a pointer to an *Inode* in the hash table using the same combination as a key. To support fast *readdir* operation, Octopus$^+$ keeps an entry list for each directory, which keeps the names of all sub-directories or files reside on this folder. Such entry lists reside in DRAM and need no persistence whatsoever, for contents can always be recovered from persistent *dentries* on data servers. Compared to its predecessor (*Octopus* [36]), such a directory tree architecture allows Octopus$^+$ to maximize the metadata scalability by reducing the overhead of multiple network round trips.

Data indexing. In Octopus$^+$, a data block has a fixed size of 4 KB. In current implementation, to simplify the management of data blocks, we do not distribute file data across multiple servers. Octopus$^+$'s 342 s `Inodes` records all data blocks of a file using a skip list. Each file owns a skip list

[1]User-space lib mode is used in evaluation to avoid the bottleneck of FUSE.



(a) Client-active Data Replication (b) Operation Log Based Metadata Replication

Fig. 7. Data and metadata replication.

instance, and this skip list resides entirely on NVM. Each node in a skip list represents a number of contiguous data blocks in the NVM pool. It has three fields: *offset* in a file, *size*, and *s_addr* that points to an NVM address. All nodes in a skip list are sorted by *s_addr* fields, and OCTOPUS+ guarantees that two neighbor nodes never overlap. By using the skip list to index data blocks, OCTOPUS+ can access a data block with a time complexity of O(logn). During the *read* operation, OCTOPUS+ searches the skip list and collects information of the certain range that a client is going to read, and sends it back to the client.

4.2 Data Availability

Failures are common issues in distributed systems. We expect OCTOPUS+ to provide high availability for both data and metadata. To this end, a certain level of redundancy is required. We build different replication mechanisms for metadata and data respectively, and by working closely with our transaction mechanism, these two together can uphold the replication feature in OCTOPUS+. We use *RF* (i.e., replication factor) to indicate the number of replicas. In our current implementation, users can set *RF* at their own need, either by setting a global *RF* that is suitable for all files or setting different *RF*s for different files. In the current implementation, we choose backup servers randomly among all servers to maintain load balance.

Metadata replication. Replicating metadata should be carried out with caution because the size of metadata to be updated is rather small in one request but the frequency can be high. As a result, directly replicating metadata among servers might be too expensive. Therefore, metadata replication should take performance into account. As shown in Figure 7(b), we thus adopt an *operation log-based replication mechanism based on RAMCloud* [46]. Specifically, when executing a file system operation that involves updating metadata, instead of propagating all persistent memory updates to backups directly, OCTOPUS+ only replicates an operation log to backup servers. An operation log entry only carries minimum information, so backup servers can replay it asynchronously in the background and bring itself up to date. Such an operation-log-based replication naturally fits into our existing *collect-dispatch transaction*: all OCTOPUS+ needs to do is to propagate the operation log to backup servers synchronously using one-sided RDMA write verbs during the local logging phase (see Figure 6), and background threads in backup servers then apply updates from the newly received logs. Normally, a file operation often involves multiple metadata updates. By introducing the operation log, OCTOPUS+ reduces network round trips by

simply synchronizing metadata updates to remote servers through a single log entry. In addition, using one-sided RDMA write further saves CPU cycles on remote servers. Note that the metadata on MDS is also duplicated to multiple servers, and we use a similar approach for replication.

Data replication. As for data replication, Octopus$^+$ extents the *client-active I/O* described in Section 3 to replicate file data to multiple servers. Specifically, during a file write operation, as shown in Figure 7(a), a client is responsible for writing data to all replicas. Note that a crash may occur when a client is replicating data to backup servers, which may lead to inconsistency issues. We address this consistency issue in the next section.

4.3 File System Consistency

In terms of the file system consistency, three issues need to be carefully handled when RDMA networks meet NVMs: (1) remote data persistency, (2) failure atomicity, and (3) concurrency control.

Data persistency. In persistent memory systems, data need to be flushed out of the volatile CPU cache and reach persistent memory in a desired order to provide crash consistency [11, 15, 37, 38, 45, 47]. However, commercial RDMA NICs do not support "remote" flush primitive since a write verb writes data to remote LLCs directly with the Data Direct I/O (DDIO) technology [20]. To still exploit the direct data access feature without compromising the data persistency requirements, we add an extra RPC after each RDMA write operation. When writing data to remote persistent memory, a client first sends an RDMA write operation using the write-with-imms verb, which carries an immediate field as an ID. Meanwhile, this client also issues an RPC, which contains the address of the written persistent memory area and the size. Once the remote server receives this immediate value, indicating that the remote write operation has been finished, it then performs a flush operation on that address on behalf of the client. In the future, we expect that hardware-based remote flush functionalities will be able to be used in such scenarios, such as remote durability [13], `RDMA commit` [53], or new designs that leverage availability for crash consistency [63].

Failure atomicity. Efforts to preserve failure atomicity in Octopus$^+$ is twofold: (1) using a copy-on-write mechanism to update file data and (2) updating metadata in a transactional manner. First, when handling file data write operations, clients never issue RDMA writes to persistent memory space that already holds valid file data. Instead, all updates are redirected to newly allocated persistent memory space. As a result, a file data block is either in its new or old state, by switching the related metadata atomically. Second, Octopus$^+$ encloses all metadata updates into a transaction for atomicity using the *collect-dispatch transaction* presented in Section 3.3.2. Since it always records new updates in *redo* logs before actually performing in-place updates, Octopus$^+$ can make sure the updates are made in an "all-or-nothing" manner. For operations that involve updating both metadata and file data (e.g., `write` and `append`), copy-on-write and the transaction mechanism are co-used to achieve the atomicity goal. Note that a client can only post metadata updating request after the data transferring has finished. In this way, Octopus$^+$ ensures that a file is switched atomically from an old state to the newest state, even in the face of system failures. In the event that a coordinator fails during a transaction, Octopus$^+$ sets one of the replica servers as the coordinator. Then the newly elected coordinator can restore the consistency of the cluster based on the state of other replica servers (e.g., the operation log). Note that this might leave the cluster short of one replica for certain files if the failed server does not come back to life.

Concurrency control. We further explain how the metadata design works with the *collect-dispatch transaction* in two concurrent scenarios. ❶ When executing a *create* operation, for example, Octopus$^+$ needs to (1) create an *Inode* in the server where the new file lands and (2) update the "number of files" field in its parent directory node (which is in the directory metadata server).

To this mission, OCTOPUS+ first starts a transaction at the server where the new file lands and acquires the lock over the parent directory node (the *Collect* phase); then OCTOPUS+ creates an *Inode* and inserts it into the hash table locally (the *Write Data* step in Figure 6); after that, OCTOPUS+ updates the "number of files" field in the parent directory node using one RDMA write and unlocks the parent directory (the *Dispatch phase*). Same as the *unlink* operation, in which the *Inode* will be removed from the hash table and recycled locally along with the data blocks. ❷ As for the *write/append* operation, since we do not allow a file to be distributed across servers, all metadata and data blocks of a file can be updated locally. As depicted in Figure 2, after the *client-active I/O* phase, OCTOPUS+ first starts a transaction at the server where a file locates and acquires the write lock for the file at the *Local Lock* phase; then updates the file's metadata locally; and last, OCTOPUS+ unlocks this file and returns to the client.

5 EVALUATION

In this section, we first evaluate OCTOPUS+'s overall data and metadata performance, then we evaluate the benefits from each mechanism design, and finally we evaluate the overall performance under macrobenchmarks.

5.1 Experimental Setup

Evaluation platform. We deploy OCTOPUS+ on servers with Intel DCPMM and RDMA networks. Each server is equipped with 192 GB of DRAM and two 2.60-GHz Intel Xeon Gold 6240M processors (36 cores per processor). For NVM configuration, each server has six 256-GB Intel Optane DCPMMs (three modules on each NUMA node). Through this entire evaluation, since cross-NUMA traffic has huge impact on performance [61], we only utilize NVMs on one NUMA node to deploy OCTOPUS+ and other file systems (i.e., only 768-GB NVMs on each server). All server machines are running Ubuntu18.04 with Linux Kernel 4.15. All client machines are running CentOS-7 with Linux Kernel 3.10. Each client server has 128 GB of DRAM and two Intel Xeon E5-2650 v4 processors. All servers and clients are equipped with MCX555A-ECAT ConnectX5 EDR HCAs (which supports 100 Gbps over InfiniBand and 100 GigE) and are connected with a Mellanox MSB7790-ES2F switch. NVM devices have asymmetric read/write bandwidth [22]; the write bandwidth is 6.7 GB/s and the read bandwidth is 20 GB/s. However the NICs have symmetric read/write performance: 12 GB/s for both read and write. Therefore, there is a "bandwidth mismatch" in our evaluation environment that needs to be clarified: for writes, NVM is the bottleneck, and for reads, the network becomes the bottleneck.

Evaluated file systems. Table 1 lists DFSs for comparison. For existing DFSs that require being deployed on top of a local file system, we build local file systems on NVM with *pmem* driver and *DAX* [4] supported in *ext4*. The EXT4-DAX [3] is optimized for NVM, which bypasses the page cache and reduces memory copies. OCTOPUS+ manages NVM spaces directly. Other file systems are allocated with full capacity of one NUMA-sided NVM. In the perspective of networks, all file systems run on RDMA directly. Specifically, memGlusterFS supports using RDMA protocol for communication between GlusterFS clients and GlusterFS bricks. NVFS is an optimized version of HDFS that is designed to better exploit the advantages of byte addressability of NVM and RDMA networks. Note that to avoid the overhead caused by *FUSE*, we conduct all evaluations on OCTOPUS+ using user-lib mode.

Workloads. We use *mdtest* for metadata evaluation, *fio* for read/write evaluation, and an in-house read/write tool based on *openMPI* for aggregated I/O performance. We use *filebench* as macrobenchmark, and choose four benchmarks from *filebench*: *Varmail*, *Fileserver*, *Webproxy*, and *Webserver*.

Table 1. Evaluated File Systems

memGlusterFS	GlusterFS runs on memory. GlusterFS is a widely used DFS that has no centralized metadata services and is now a part of Redhat.
NVFS [21]	A version of HDFS that is optimized with both RDMA and NVM.

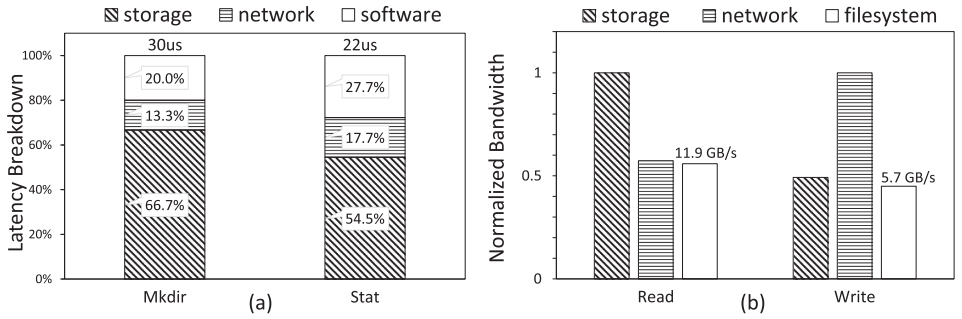


Fig. 8. Latency breakdown and bandwidth utilization.

5.2 Overall Performance

In this evaluation, we first compare OCTOPUS$^+$'s latency and bandwidth to the raw network's and storage's latency and bandwidth, then compare OCTOPUS$^+$'s metadata and data performance to other file systems using *mdtest* and *fio*.

5.2.1 Latency and Bandwidth Breakdown. Figure 8 shows latency and bandwidth breakdown for OCTOPUS$^+$. All directories and files are created in the root directory. From Figure 8, we have two observations.

First, the software latency is dramatically reduced. For example, in *Mkdir*, the software latency is 6.3 us (around 20% of the total latency) in OCTOPUS$^+$, from 363 us (over 99%) in memGlusterFS, as shown in Figure 8(a). For memGlusterFS on the emerging NVM and RDMA hardwares, the file system layer produces a latency that is several orders larger than that of the storage or the network. The software consumes the overwhelmed part and becomes a new bottleneck of the whole storage system. In contrast, OCTOPUS$^+$ is effective in reducing the software latency by redesigning the data and metadata mechanisms with RDMA and NVM. The software latency in OCTOPUS$^+$ is in the same order with the hardware.

Second, OCTOPUS$^+$ achieves read/write bandwidth that further approaches the raw hardware bandwidth. The raw storage and network bandwidths respectively are, for read operations, 20 GB/s (with multi-thread memcpy) and 12.3 GB/s, and for write operations, 6.3 GB/s (with multi-thread memcpy) and 12.3 GB/s. As shown in Figure 8(b), OCTOPUS$^+$ achieves a read/write (11.9/5.7 GB/s) bandwidth that is 59%/90% of the NVM bandwidth and is 99%/46% of the network bandwidth (for write operations, the network bandwidth is much higher than that of NVM). In conclusion, OCTOPUS$^+$ can effectively exploit the hardware performance.

5.2.2 Metadata Performance. In this evaluation, we run *mdtest* to evaluate the performance of metadata operations. In such process, clients create, access, and remove directories and files in a multi-level fashion. We then measure the average throughput of each metadata operation. Figure 9 shows the file systems' average performance in terms of metadata IOPS by varying the number of servers. From Figure 9, we make two observations.

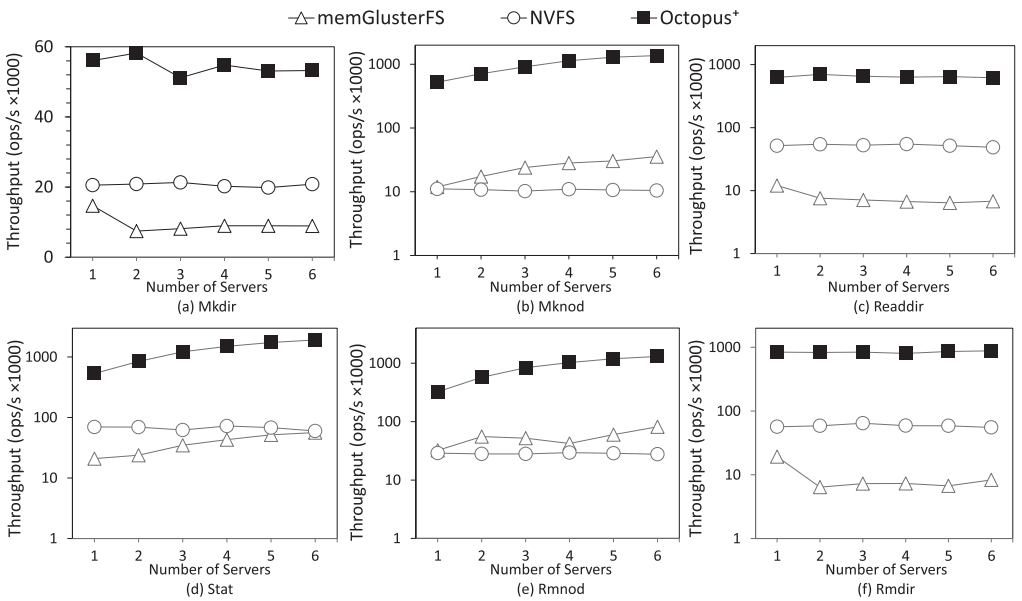


Fig. 9. Metadata throughput.

First, Octopus+ has the highest metadata IOPS among all evaluated file systems in general. As shown in Figure 9, both memGlusterFS and NVFS provide metadata IOPS in the order of 10^4. Comparatively, Octopus+ provides metadata IOPS at least in the order of 10^5 except for *Mkdir*. For *Stat Mknod* and *Unlink*, Octopus+ provides IOPS in the order of 10^6, two orders higher than that of memGlusterFS and NVFS. Generally, Octopus+ achieves high throughput in processing metadata requests, which mainly owes to the *self-identified RPC*. The *self-identified RPC* promises extremely low latency and high throughput. However, for memGlusterFS and NVFS, because of their dependence on local file system and their separate network and storage architecture, they both suffer from heavy software overheads and therefore miss the opportunity to fully exploit hardware performance.

Second, for file operations, Octopus+ achieves good scalability, and for directory operations, Octopus+ shows stable performance and still outperforms the others. Octopus+ shows good scalability in file operations for two reasons. First, the RPC and transaction designs have good scalability. Second, files are distributed to multiple servers in a hash-based fashion, which is good for scalability. As for directory operations, Octopus+ shows stable performance because of the centralized directory metadata server design. However, as shown in Figure 9, from one server to six servers, NVFS hardly scales at all. NVFS is designed with single metadata server, and all metadata operations have to go through this server; therefore, the sole metadata server becomes the bottleneck. MemGlusterFS also performs poorly in terms of metadata IOPS, because GlusterFS is designed to run on hard disks and the software layer is inefficient in exploring the high performance of NVM and RDMA, which has been illustrated in Section 2.2. We also noticed that memGlusterFS performs extremely poor in directory operations. This is because GlusterFS deploys a decentralized metadata architecture by creating the same directory on multiple servers, which is not friendly to directory operations. We notice that *Mkdir* has relatively low performance compared to other operations in Octopus+. This is because, in the current implementation, *Mkdir* requires more NVM accesses and has to be serialized.

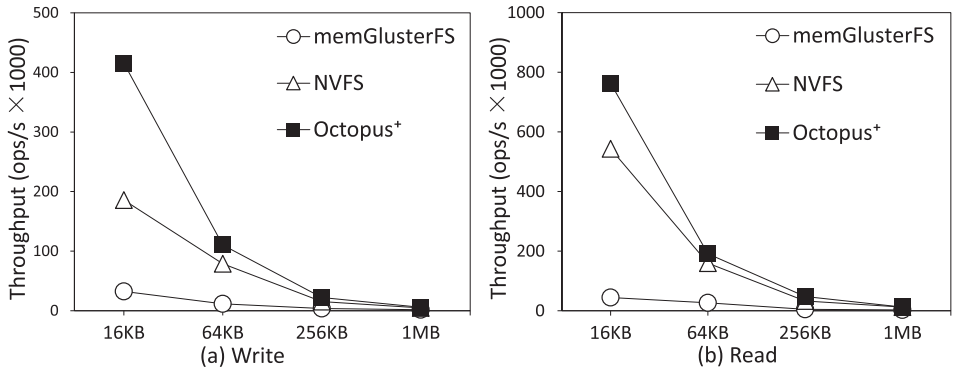


Fig. 10. Data I/O throughput (multiple clients and single server).

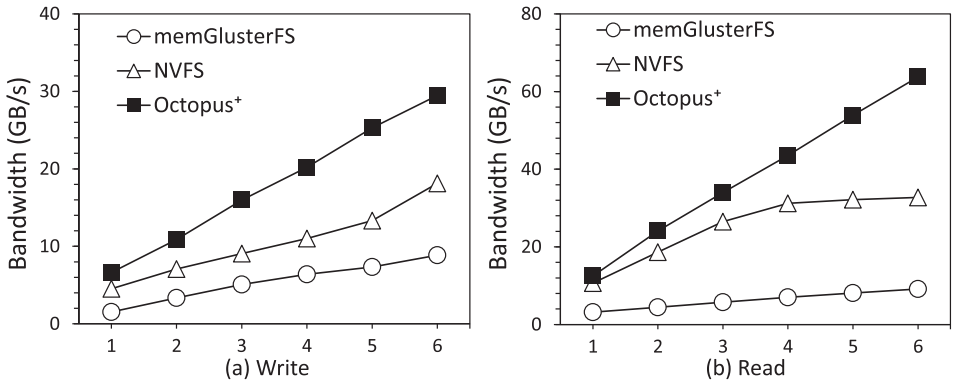


Fig. 11. Data I/O bandwidth (multiple clients and multiple servers).

5.2.3 Read/Write Performance. In this evaluation, we use *fio* to measure the read/write performance of file systems. All clients operate on their own file. Figure 10 shows the file systems' performance in terms of concurrent read/write throughput with multiple clients and a single server by varying the I/O sizes. From Figure 10, we observe that, with relative small read/write sizes, OCTOPUS+ achieves much higher throughput than other file systems. Taking I/O size of 16 KB as an example, OCTOPUS+ achieves 415 Kops/s and 763 Kops/s for write and read, respectively, which is 2.2 times and 1.4 times of NVFS's performance, respectively. As I/O size increases, OCTOPUS+ 523 still shows better throughput than NVFS and memGlusterFS. This benefit is a combined effect from the *client-active data I/O*, *self-identified RPC*, and metadata updating mechanisms using *collect-dispatch transaction*. However, NVFS and memGlusterFS both suffer from overwhelmed software overheads rooted in their inefficient layered software designs and fail to push performance any further to hardware capability. For example, memGlusterFS only achieves a throughput at 32 Kops/s for I/O size of 16 KB, and even when I/O size is at 1 MB, memGlusterFS only achieves a throughput at 1,529 op/s, which is only a small piece of hardware performance.

We then evaluated the aggregated bandwidth. Figure 11 shows the read/write bandwidth achieved by the cluster with multiple clients and multiple servers. The procedure is the same as tha presented earlier. As shown in Figure 11, as the server increases from one to six, OCTOPUS+ scales smoothly, and significantly outperforms other DFSs in terms of read and write bandwidth. For read performance, OCTOPUS+ provides an aggregated bandwidth that is close to the network

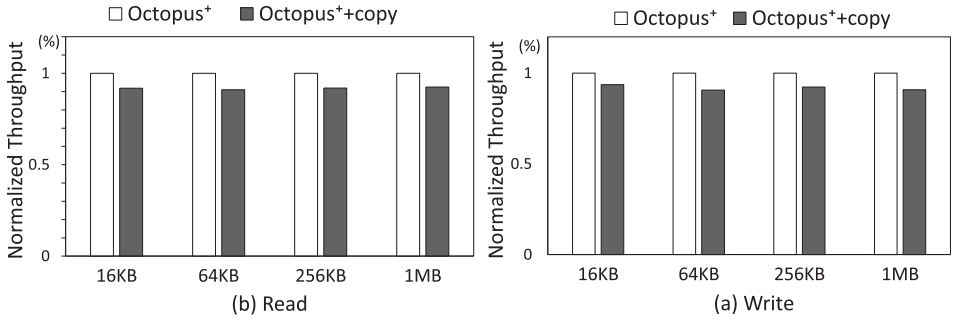


Fig. 12. Effects of reducing data copies.

hardware performance. Although both NVFS and memGlusterFS's read/write bandwidths increase when there are more servers in the cluster, the aggregated bandwidth only approaches half of the raw hardware bandwidth. For example, even with six servers in the cluster, memGlusterFS only achieves 8,867 MB/s write bandwidth and 9,156 MB/s read bandwidth. Both memGlusterFS and NVFS fail to fully exploit hardware bandwidth because they both rely on local file systems and have inefficient software designs. In contrast, by introducing a shared persistent memory pool to reduce data copies and actively performing I/Os in clients, Octopus+ achieves bandwidth close to that of the raw hardware performance.

5.3 Evaluation of Internal Mechanisms

In this section, we evaluate the effects of each internal mechanism. We first evaluate the effects of reducing data copies, then we evaluate the effects of *self-identified metadata RPC* and the effects of the replication mechanism.

5.3.1 Effects of Reducing Data Copies. Octopus+ improves data transfer bandwidth by reducing memory copies. To verify the benefits, we implement a version of Octopus+ that adds extra data copies at the client side, and we refer to it as the *Octopus+ +copy*. In this evaluation, we launch multiple clients to send file write requests to a single server. As shown in Figure 12, Octopus+ achieves more IOPS than *Octopus+ +copy*. In fact, up to 9% of total throughput can be gained when cutting down extra data copies. We believe that this mechanism may profit more in larger scales.

5.3.2 Effects of Self-Identified Metadata RPC. To evaluate the effects of Octopus+'s self-identified metadata RPC, we first compare the raw RPC performance of Octopus+ with other RPC frameworks, then we compare Octopus+'s metadata latency with existing file systems. Both evaluations are carried using in-house microbenchmarks implemented using *openMPI*. We choose three RPC frameworks: *eRPC* [25], *DaRPC* [51], and *LITE* [54]. We turn off the batching feature for all RPC frameworks to ensure a fair comparison. Figure 13(a) shows the raw RPC throughput using four RPC frameworks (i.e., DaRPC; eRPC; LITE; and self-identified, which is used in Octopus+) given different I/O sizes.

Since *DaRPC* is designed based on RDMA `send/recv`, and `send/recv` have lower throughput than `memory` verbs, it achieves the lowest throughput, at 2.6 Mops/s with an I/O size of 16 bytes. Its performance may also be limited by the jVerbs interface. LITE, *eRPC*, and self-identified RPC are based on `memory` verbs, and therefore they have higher throughput. Our proposed self-identified RPC, which carries on client identifiers with the RDMA `write_with_imm` verbs, achieves the highest throughput, at 4.1 Mops/s when I/O size is set to 16 bytes, and 3.4 Mops/s when I/O size is

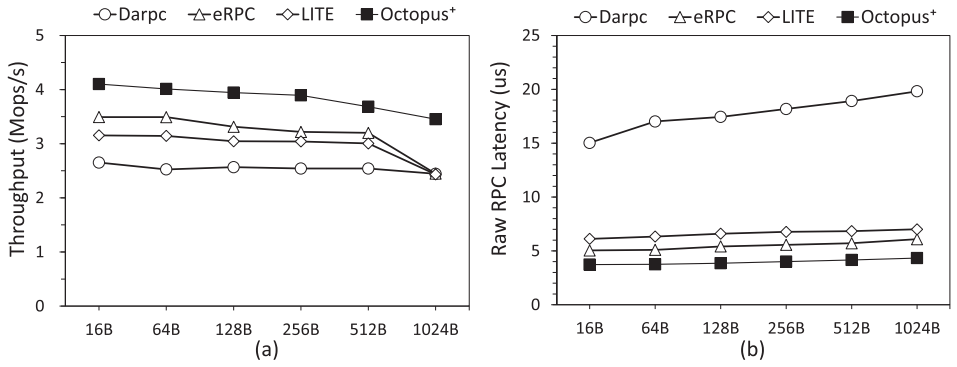


Fig. 13. Raw RPC performance.

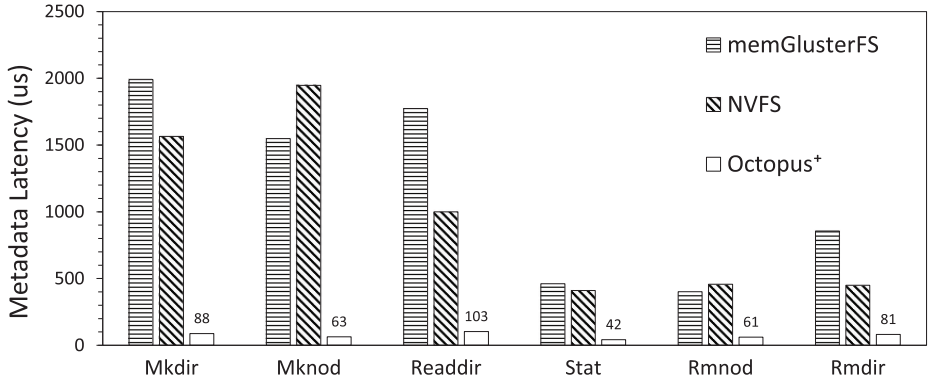


Fig. 14. Metadata latency.

1 MB. Similarly, we also measure the latency of each RPC by varying the I/O size (in Figure 13(b)). Our *self-identified RPC* keeps relative low latency, which is suitable for distributed storage systems to support a large number of client requests. *LITE* works in kernel space and interacts with applications through *syscall*, during which the *context switch* brings in extra overheads, and therefore *LITE* has more latency than *eRPC* and *self-identified RPC*.

From these results, we observe that *eRPC* has performance comparable to *self-identified RPC*, and the higher latency may be because *eRPC* has a more complex software stack. In Octopus$^+$, we simplify the RPC implementation and make it simply suffice to work for our file system.

To evaluate metadata latency, we create directories in a multi-level fashion and fill them with multiple regular files. Figure 14 shows the average metadata latency of Octopus$^+$ along with other file systems.

As shown in Figure 14, Octopus$^+$ achieves the lowest metadata latency among all of the evaluated file systems for all evaluated metadata operations (e.g., 63 us and 103 us respectively for `Mknod` and `Readdir`), which is extremely lower than other file systems. With the *self-identified metadata RPC* and the efficient internal metadata design, Octopus$^+$ can support low-latency metadata operations even without client cache. In contrast, NVFS and memGlusterFS have much higher metadata latency (e.g., memGlusterFS needs 1,990 us to create a directory, and NVFS spends 1,945, us to create files). The reason behind this is that their heavy file system designs and inefficient network mechanisms together produce huge software overheads. For memGlusterFS, it is the

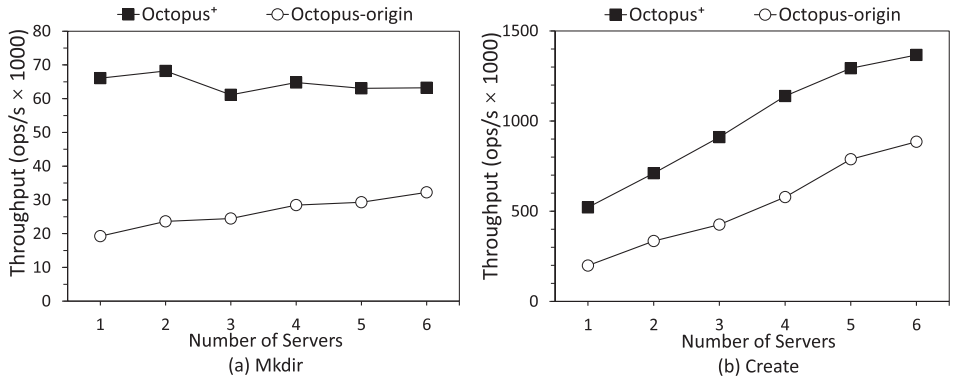


Fig. 15. Metadata performance under different directory tree designs.

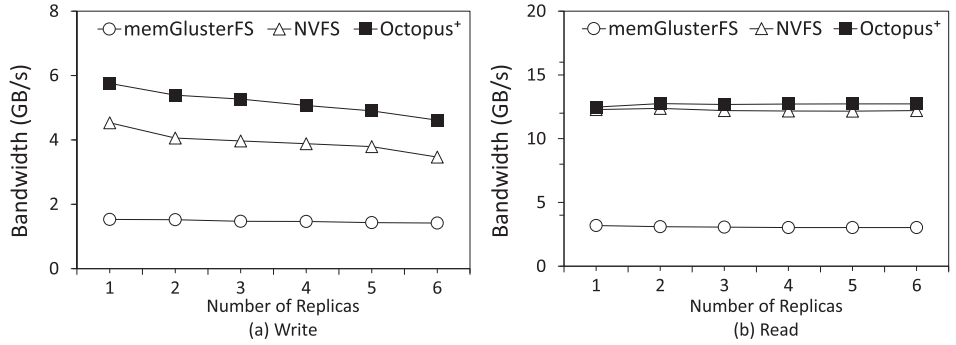


Fig. 16. Bandwidth under different replication factors.

stacked software architecture inherited from GlusterFS, and for NVFS, the case is the framework of HDFS.

5.3.3 Effects of the Directory Tree Design. To evaluate the effects of the new directory tree design over the original hash-based design in *Octopus*, we evaluate the metadata throughput in *mkdir* and *create*. We use the same benchmarks used in Section 5.2.2 to build a multi-level directory tree. We then measure the throughput of both directory tree designs as the number of servers increases. Figure 15 shows the throughput of both directory tree designs. The original hash-based directory tree design in *Octopus* is denoted as "*Octopus-origin*." From the results, we observe that the new directory tree design can deliver much better metadata performance than the previous one. In fact, OCTOPUS+ achieves throughputs several times higher than *Octopus-origin*. This is mostly due to the reduction in network round trips.

5.3.4 Effects of the Replication Mechanism. To evaluate the effects of the replication mechanism in OCTOPUS+, we measure the aggregate read/write bandwidth provided by the cluster with different replication factors. We conduct the same procedure used to evaluate read/write performance in Section 5.2.3. Figure 16 exhibits the read/write performance of three file systems under different replication factors. We observe that OCTOPUS+ achieves the best performance among all evaluated file systems. For the write operation, the replication factor has more prominent impact on aggregate bandwidth, and as the replication factor increases, the performance drops quickly for all file systems. Still, OCTOPUS+ outperforms other file systems on account of the *client-active*

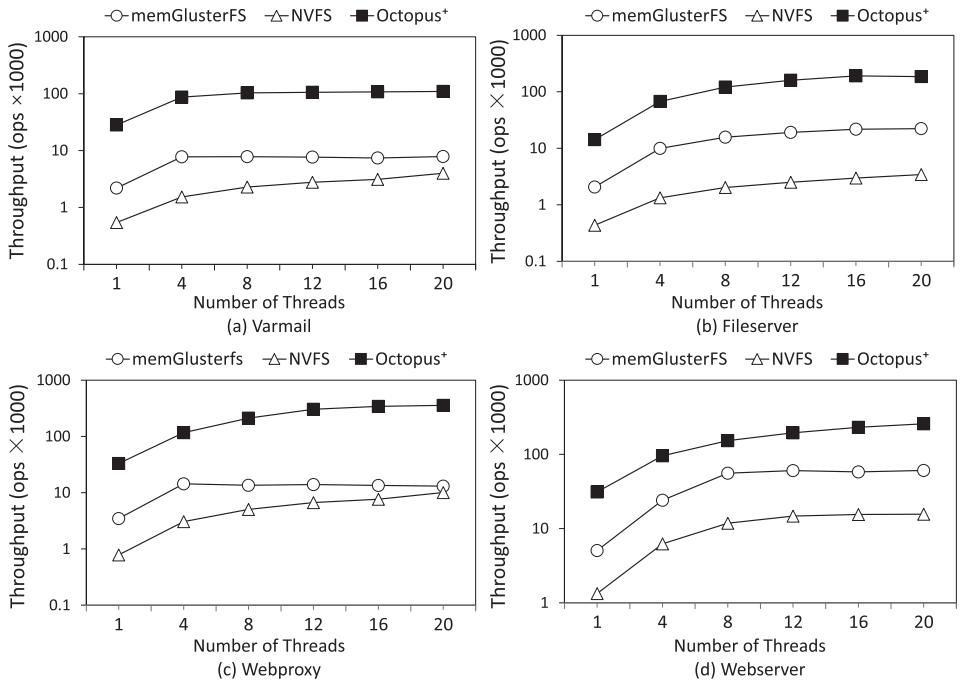


Fig. 17. Filebench evaluation.

replication mechanism that alleviates the extra pressure on server side. For the read operation, the setback has reduced a lot, and NVFS and Octopus$^+$ have comparable performance. The data replication mechanism hardly bothers read bandwidth because the clients only interact with the primary node to read data.

5.4 Evaluation Using Macrobenchmarks

In addition, we compare Octopus$^+$ with other DFSs under benchmarks from *filebench*.

Overall performance under filebench. We select four benchmarks from *filebench* (i.e., *varmail*, *fileserver*, *webproxy*, and *webserver*). In this evaluation, clients launch a series of both metadata and data file operations to a bunch of files. The cluster is configured with one server, and the replication feature is disabled. Since NVFS is developed based on the HDFS framework with non-posix interfaces, it cannot interact with *filebench* directly. To overcome this obstacle, we implement *filebench* using interfaces of NVFS from the hdfs library (i.e., libhdfs). Such modification results in two kinds of additional software overheads. The first one is software overhead caused when communicating with NVFS through JNI and the hdfs library. The second one is that, even with the hdfs library, the HDFS framework still lacks posix-compatible semantics. As a result, certain file operations in *filebench* end up executing two interfaces from libhdfs. Figure 17 shows the overall performance for different file systems under different benchmarks by varying the number of client threads. We can see that as the number of client threads increases, the performance curve presents a plateau eventually. We believe at that point the single client server reaches saturation. NVFS achieves the worst performance for reasons mentioned earlier. However, Octopus$^+$ achieves good performance and scalability. As shown in Figure 17, Octopus$^+$ achieves at most two orders

of magnitude of performance speedup. The results suggest that all of the mechanisms described before can work effectively under macrobenchmarks.

6 RELATED WORK

Persistent memory data structures. Persistent memories have both persistence and byte-addressability benefits, which are perfect to support persistent data structure [18, 32]. FAST&FAIR [18] proposes Failure-Atomic Shift (FAST) and Failure-Atomic In-place Rebalance (FAIR) algorithms and builds B+-trees in a byte-addressable fashion. RECIEPE [11] proposes a principled approach that can convert concurrent DRAM indexes into crash-consistent indexes for persistent memory, providing new insight in developing persistent indexes. OCTOPUS+ draws lessons from them in developing data structures for persistent memories and use it in building data structures for the file system.

Persistent memory file systems. In addition to file systems that are built for flash memory [23, 31, 39, 40, 62], a number of local file systems have been built from scratch to exploit both byte addressability and persistence benefits of NVM [11, 12, 15, 24, 29, 45, 57–59, 64]. BPFS [11] is a file system for persistent memory that directly manages NVM in a tree structure and provides atomic data persistence using short-circuit shadow paging. PMFS [15] proposed by Intel also enables direct persistent memory access from applications by removing the file system page cache with memory mapped I/O. Similar to BPFS and PMFS, SCMFS [57] is a file system for persistent memory that leverages the virtual memory management of the operating system. Fine-grained management is further studied in the recent NOVA [58] and HiNFS [45] to make software more efficient. The Linux kernel community has also started to support persistent memory by introducing DAX (Direct Access) to existing file systems (e.g., EXT4-DAX [3]). NOVA-Fortis [59] is developed based on NOVA [58], and it enables NOVA with a snapshot feature to improve file system reliability. Strata [29] and Ziggurat [64] try to design the NVM file system in the context of a tied storage system. SplitFS [24] and ZoFS [12] both propose to further reduce software overhead by exporting the file system to user space. SplitFS proposes a split of responsibilities between a user-space library file system and an existing kernel PM file system. ZoFS tackles the protection problem of the user-space file system and proposes a new abstraction called *coffer* to help build a user-space NVM file system architecture that allows direct management over NVM resources in user space while providing protection and isolation.

The efficient software design concept in these local file systems, including removing duplicated memory copies, is further studied in the OCTOPUS+ DFS to make remote accesses more efficient.

General RDMA optimizations. RDMA provides high performance but requires careful tuning. A recent study [27] offers guidelines on how to use RDMA verbs efficiently from a low-level perspective such as in PCIe and NIC. Cell [43] dynamically balances CPU consumption and network overhead using RDMA primitives in a distributed B-tree store. PASTE [17] proposes direct NIC DMA to persistent memory to avoid data copies, for a joint optimization between network and data stores. FaSST [28] proposes to use UD for RPC implementation when using `send/recv`, to improve scalability. RDMA has also been used to optimize distributed protocols, like shared memory access [14], replication [63], in-memory transaction [56], and lock mechanism [44]. RDMA optimizations have brought benefits to computer systems, and this motivates us to start rethinking the file system design with RDMA.

RDMA optimizations in key-value stores. RDMA features have been adopted in several key-value stores to improve performance [10, 14, 26, 41, 42, 55]. MICA [35] bypasses the kernel and uses a lightweight networking stack to improve data access performance in key-value stores. Pilaf [42] optimizes the *get* operation using multiple RDMA `read` commands at the client side, which

offloads the hash calculation burden from remote servers to clients, improving system performance. HERD [26] implements both *get* and *put* operations using the combination of RDMA `write` and UD `send`, to achieve high throughput. HydraDB [55] is a versatile *key-value* middleware that achieves data replication to guarantee fault tolerance and awareness for the NUMA architecture, and adds a client-side cache to accelerate the *get* operation. FlatStore [10] is a PM-based key-value storage engine, which decouples the role of a key-value store into a persistent log structure for efficient storage and a volatile index for fast indexing. AsymNVM [41] proposes a disaggregation architecture that is upheld by high-performance RDMA networks. In this architecture, NVM devices can be shared by multiple servers and provide recoverable persistent data structures.

Although RDMA techniques lead to evaluations in the designs of key-value stores, their impact on file system designs is still underexploited.

RDMA optimizations in DFSs. Existing DFSs have tried to support the RDMA network by substituting their communication modules [2, 5, 16]. Ceph over Accelio [2] is a project under development to support RDMA in Ceph. Accelio [1] is an RDMA-based asynchronous messaging and RPC middleware designed to improve message performance and CPU parallelism. Alluxio [33] in Spark (formerly named *Tachyon*) is transplanted to run on top of RDMA by Mellanox [5]. It faces the same problem as Ceph on RDMA. NVFS [21] is an optimized version of HDFS that combines both NVM and RDMA technologies. Due to heavy software design in HDFS, NVFS hardly exploits the high performance of NVM and RDMA. Crail [7] is a recently developed distributed in-memory file system built on DaRPC [51]. DaRPC is an RDMA-based RPC that tightly integrates the RPC message processing and network processing, which provides both high throughput and low latency. However, their internal file system mechanisms remain the same. In comparison, our proposed Octopus$^+$ revisits the file system mechanisms with RDMA features, instead of introducing RDMA only to the communication module. Recent works(e.g., [8, 49, 60]) propose new file system designs. The remote region [8] exports part of a process's memory as files, and accesses them through file system operations over RDMA networks. The remote region only uses the file system interface to access remote memory, which is different from Octopus$^+$'s goal. Hotpot [49] and Orion [60] both introduce a noble DFS for NVM, and in contrast to Octopus$^+$, they both work in kernel space; they also have a different metadata layout and a different consistency model.

7 CONCLUSION

The efficiency of file system design becomes an important design issue for storage systems that are equipped with high-speed NVM and RDMA hardware. Both of the two emerging hardware technologies not only improve hardware performance but also push forward software evolution. In this article, we propose a distributed memory file system, Octopus$^+$, which has its internal file system mechanisms closely coupled with RDMA features. Octopus$^+$ simplifies the data management layer by reducing memory copies, and rebalances network and server loads with active I/Os in clients. It also redesigns the metadata RPC and the distributed transaction by using RDMA primitives. Evaluations on real NVM devices and RDMA devices show that Octopus$^+$ effectively explores hardware benefits and significantly outperforms existing DFSs.

REFERENCES

[1] NVIDIA. 2013. Accelio. Retrieved June 20, 2021 from https://github.com/accelio/accelio.
[2] CohortFS. 2014. Ceph over Accelio. Retrieved June 20, 2021 from https://www.cohortfs.com/sites/default/files/ceph%20day-boston-2014-06-10-matt-benjamin-cohortfs-mlx-xio-v5ez.pdf.
[3] LWN.net. 2014. Support ext4 on NV-DIMMs. Retrieved June 20, 2021 from https://lwn.net/Articles/588218.
[4] LWN.net. 2014. Supporting Filesystems in Persistent Memory. Retrieved June 20, 2021 from https://lwn.net/Articles/610174.

[5] Mellanox. 2015. RDMA Improves Alluxio (Tachyon) Remote Read Bandwidth and CPU Utilization by up to 50%. Retrieved June 20, 2021 from https://community.mellanox.com/docs/DOC-2128.

[6] SAP HANA. 2016. In-Memory Computing and Real Time Analytics. Retrieved June 20, 2021 from https://www.sap.com/products/hana.html.

[7] GitHub. 2017. Crail: A Fast Multi-Tiered Distributed Direct Access File System. Retrieved June 20, 2021 from https://github.com/zrlio/crail.

[8] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, et al. 2018. Remote regions: A simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 775–787. https://www.usenix.org/conference/atc18/presentation/aguilera.

[9] InfiniBand Trade Association. 2009. *InfiniBand Architecture Specification: Release 1.3*. InfiniBand Trade Association.

[10] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. ACM, New York, NY, 1077–1091. https://doi.org/10.1145/3373376.3378515

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

[12] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. ACM, New York, NY, 478–493. https://doi.org/10.1145/3341301.3359637

[13] Chet Douglas. 2015. RDMA with PMEM: Software mechanisms for enabling access to remote persistent memory. Retrieved June 20, 2021 from http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 401–414.

[15] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY, Article 15, 15 pages.

[16] Gluster. 2020. GlusterFS on RDMA. Retrieved June 20, 2021 from https://gluster.readthedocs.io/en/latest/AdministratorGuide/RDMATransport/.

[17] Michio Honda, Lars Eggert, and Douglas Santry. 2016. PASTE: Network stacks must integrate with NVMM abstractions. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, New York, NY, 183–189.

[18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang.

[19] Intel. 2019. Intel Optane DC Persistent Memory. Retrieved June 20, 2021 from https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[20] Intel. 2020. Intel Data Direct I/O Technology. Retrieved June 20, 2021 from https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[21] Nusrat Sharmin Islam, Md Wasi-Ur Rahman, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. High performance design for HDFS with byte-addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, New York, NY, 8.

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, et al. 2019. Basic performance measurements of the Intel Optane DC persistent memory module. arXiv:1903.05714.

[23] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. 2010. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*.

[24] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. ACM, New York, NY, 494–508. https://doi.org/10.1145/3341301.3359631

[25] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 1–16.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. 295–306.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 185–201.

[29] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 460–477. https://doi.org/10.1145/3132747.3132770

[30] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 2–13.

[31] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[32] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. ACM, New York, NY, 462–477. https://doi.org/10.1145/3341301.3359635

[33] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*.

[34] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. Locofs: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. 1–12.

[35] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. *Management* 15, 32 (2014), 36.

[36] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: An RDMA-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu.

[37] Youyou Lu, Jiwu Shu, and Long Sun. 2015. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st Conference on Massive Storage Systems and Technologies (MSST'15)*. IEEE, Los Alamitos, CA, 1–13.

[38] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-ordering consistency for persistent memory. In *Proceedings of the IEEE 32nd International Conference on Computer Design (ICCD'14)*. IEEE, Los Alamitos, CA.

[39] Youyou Lu, Jiwu Shu, and Wei Wang. 2014. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 75–88.

[40] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*.

[41] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. ACM, New York, NY, 757–773. https://doi.org/10.1145/3373376.3378511

[42] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. 103–114.

[43] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and network in the cell distributed B-tree store. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*.

[44] Sundeep Narravula, A. Marnidala, Abhinav Vishnu, Karthikeyan Vaidyanathan, and Dhabaleswar K. Panda. 2007. High performance distributed lock management services using network-based remote atomic operations. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*. IEEE, Los Alamitos, CA, 583–590.

[45] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A high performance file system for non-volatile main memory. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, New York, NY, 12.

[46] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, et al. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems* 33, 3 (2015), 1–55.

[47] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA'14)*. 265–276.

[48] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 24–33.

[49] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. ACM, New York, NY, 323–337. https://doi.org/10.1145/3127479.3128610

[50] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80–83.

[51] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. 2014. DaRPC: Data center RPC. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*. ACM, New York, NY, 1–13.

[52] Steven Swanson and Adrian M. Caulfield. 2013. Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage. *Computer* 46, 8 (2013), 52–59.

[53] Tom Talpey. 2015. Remote Access to Ultra-Low-Latency Storage. Retrieved June 20, 2021 from http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/Talpey-Remote_Access_Storage.pdf.

[54] Shin-Yeh Tsai and Yiying Zhang. 2017. Lite kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 306–324.

[55] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. HydraDB: A resilient RDMA-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM, New York, NY, 22.

[56] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, New York, NY, 87–104.

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

[58] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 323–338.

[59] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 478–496. https://doi.org/10.1145/3132747.3132761

[60] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 221–234. https://www.usenix.org/conference/fast19/presentation/yang.

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

[62] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: A log-structured file system to exploit the internal parallelism of flash devices. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*.

[63] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 3–18. https://doi.org/10.1145/2694344.2694370

[64] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A tiered file system for non-volatile main memories and disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 207–219. https://www.usenix.org/conference/fast19/presentation/zheng.

[65] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 14–23.

Received July 2020; revised November 2020; accepted January 2021