Octopus: an RDMA-enabled Distributed Persistent Memory File System

Youyou Lu, Jiwu Shu, and Youmin Chen, *Tsinghua University;* Tao Li, *University of Florida*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Octopus: an RDMA-enabled Distributed Persistent Memory File System

Youyou Lu
Tsinghua University

Jiwu Shu*
Tsinghua University

Youmin Chen
Tsinghua University

Tao Li
University of Florida

Abstract

Non-volatile memory (NVM) 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 paper, we propose an RDMA-enabled distributed persistent memory file system, Octopus, to redesign file system internal mechanisms by closely coupling NVM 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 re-balance the load between the server and network. For metadata operations, Octopus introduces self-identified RPC for immediate notification between file systems and networking, and an efficient distributed transaction mechanism for consistency. Evaluations show that Octopus achieves nearly the raw bandwidth for large I/Os and orders of magnitude better performance than existing distributed file systems.

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 [8]) and file systems (e.g., Alluxio [23]), have been used to achieve high data processing performance. With the emerging non-volatile memory (NVM) technologies, such as phase change memory (PCM) [34, 21, 46], resistive RAM (ReRAM), and 3D XPoint [7], data can be stored persistently in main memory level, i.e., *persistent memory*. New local file systems, including BPFS [11], SCMFS [42], PMFS [14], and HiNFS [32], are built recently to exploit the byte-addressability or persistence advantages of non-volatile memories. Their promising results have shown potentials of NVMs in high performance of both data storage and processing.

*Jiwu Shu is the corresponding author.

Meanwhile, the 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 $0.9us$ and $6.35GB/s$ with a 56 Gbps InfiniBand switch, compared to $75us$ and $118MB/s$ with Gigabit Ethernet (GigaE). RDMA has greatly improved data center communications or RPCs in recent studies [13, 37, 19, 20].

Distributed file systems are trying to support RDMA networks for high performance, but mostly by substituting the communication module with an RDMA library. CephFS supports RDMA by using Accelio [2], an RDMA-based asynchronous RPC middleware. GlusterFS implements its own RDMA library for data communication [1]. NVFS [16] is a HDFS variant that is optimized with NVM and RDMA. And, Crail [9], a recent distributed file system from IBM, is built on the RDMA-optimized RPC library, DaRPC [37]. 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, while it is only 2% on disk. Similarly, it achieves only 15% of raw InfiniBand bandwidth, compared to 70% of the GigaE bandwidth. In conclusion, the *strict isolation* between the file system and network layers makes distributed file systems too heavy to exploit the benefits of emerging high-speed hardware.

In this paper, we revisit both data and metadata mechanism designs of the distributed file system by taking NVM and RDMA features into consideration. We propose an efficient distributed persistent memory file system, Octopus[1], 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, avoiding stacking a distributed file system layer on local file systems, to eliminate 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 which carries 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. 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, and a collect-dispatch distributed transaction for low-overhead consistency.
- We implement and evaluate Octopus. Experimental results show that Octopus effectively explores the raw hardware performance, and significantly outperforms existing RDMA-optimized distributed file systems.

2 Background and Motivation

2.1 Non-volatile Memory and RDMA

Non-Volatile Memory. Byte-addressable non-volatile memory (NVM) technologies, including PCM [34, 21, 46], ReRAM, Memristor [36], are being intensively studied in recent years. Intel and Micron have announced the 3D XPoint technology which is expected to be in product in the near future [7]. These NVMs have access latencies close to that of DRAM, while providing data persistence as hard disks. In addition, NVMs are expected to have better scalability than DRAM [34, 21]. Therefore, NVMs are promising candidates for storing data persistently at the main memory level.

[1]It is called Octopus because the file system performs remote direct memory access just like a Octopus uses its eight legs.

Remote Direct Memory Access. 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 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 NVMs and RDMA enabled network, the hardware provides extremely higher performance than traditional media like hard disks and Gigabit Ethernet. Comparatively, overheads of the software layer, which are negligible compared to slow disk and Ethernet, now account for a significant part in the whole system.

Latency. To understand the latency overhead of existing distributed file systems, we perform synchronous 1KB write operations on GlusterFS, and collect latencies respectively in the storage, network, and software parts. The latencies are averaged with 100 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 [4], which is optimized for NVM by bypassing the page cache and reducing memory copies. In diskGluster, the storage latency consumes the most part, nearly 98% of the total latency. In memGluster, the storage latency percentage drops dramatically to nearly zero. In comparison, the file system software latency becomes the dominate part, almost 100%. Similar trends have also been observed in previous studies in local storage systems [38]. While most distributed file systems stack the distributed data

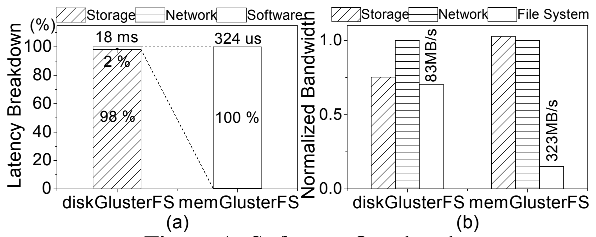

Figure 1: Software Overhead

management layer on another local file system (a.k.a, stacked file system layers), they face more serious 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 1MB 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 93.6% of raw disk bandwidth and 70.3% of raw Gigabit Ethernet bandwidth. In memGluster, GlusterFS's bandwidth is only 14.7% of raw memory bandwidth and 15.1% 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 distributed file systems. First, data are copied multiple times in multiple places in memory, including user buffer, file system page cache, and network buffer. While 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 be easily 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, distributed file systems 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. 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*.

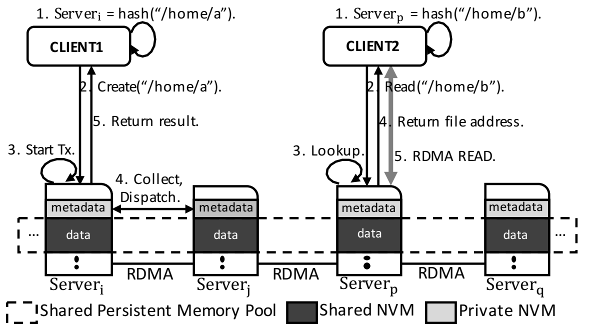

Figure 2: Octopus Architecture

3.1 Overview

Octopus is built for a cluster of servers that are equipped with non-volatile memory and RDMA-enabled networks. Octopus consists of two parts: *clients* and *data servers*. Octopus has no centralized metadata server, and the metadata service is distributed to different data servers. In Octopus, files are distributed to data servers in a hash-based way, as shown in Figure 2. A file has its metadata and data blocks in the same data server. But its parent directory and its siblings may be distributed to other servers. Note that the hash-based distribution of file or data blocks is not a design focus of this paper. Hash-based distribution may lead to difficulties in wear leveling issue in non-volatile memory, and we leave this problem for future work. Instead, we aim to discuss novel metadata and data mechanism designs that are enabled by RDMA in this paper.

In each server, the data area is exported and shared in the whole cluster for remote direct data accesses, while the metadata area is kept private for consistency reasons. Figure 3 shows the data layout of each server, which is organized into six zones: (1) *Super Block* to keep the metadata of the file system. (2) *Message Pool* for the metadata RPC for temporary message storage when exchanging messages. (3) *Metadata Index Zone* using a chained hash table to index the file or directory metadata nodes in the metadata zone. Each entry in the chained hash table contains `name`, `i_addr`, and `list_ptr` fields, which respectively represent the name of the file, the physical address of the file's inode, and the pointer to link the metadata index for the files that has a same hash value. A file hashes its name and locates its metadata index to fetch its `inode` address. (4) *Metadata Zone* to keep the file or directory metadata nodes (i.e., `inode`), each of which consumes 256 bytes. With the

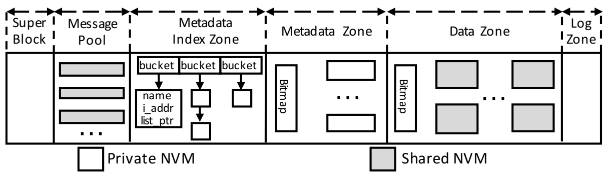

Figure 3: Data Layout in a Octopus Node

`inode`, Octopus locates the data blocks in the data zone. (5) *Data Zone* to keep data blocks, including directory entry blocks and file data blocks. (6) *Log Zone* for transaction log blocks to ensure file system consistency.

While a data server keeps metadata and data respectively in the private and shared area, Octopus accesses the two areas remotely in different ways. For the private metadata accesses, Octopus uses optimized remote procedure calls (RPC) as in existing distributed file systems. 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 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).

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 distributed file systems, a distributed file system 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

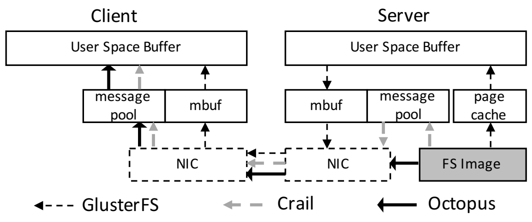

Figure 4: Data Copies in a Remote I/O Request

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 NIC (network interface controller) for final access.

Recent local persistent file systems (like PMFS [14] and EXT4-DAX [4]) directly access persistent memory storage without going through kernel page cache, but it does not solve problems in the distributed file systems cases. With direct access of these persistent memory file systems, only page cache is bypassed, and a distributed file system 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 [9], a recent RDMA-aware distributed file system built from scratch. Compared to stacked file system designs like GlusterFS, data copies in Octopus and Crail do not need to go through user space buffer in the server side, as shown in Figure 4.

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 non-volatile memory 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 non-volatile memories, 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.

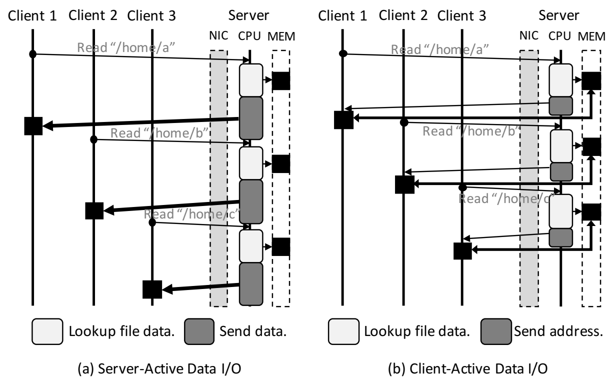

Figure 5: Comparison of Server-Active and Client-Active Modes

Similarly, a write request can also complete with one round-trip. This is called *Server-Active Mode*. While 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 1KB writes, but have to spend around $2us$ (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 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 RDMA `read` and `write` are one-sided operations, which access remote data without participation of CPUs in remote servers, 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, server load is offloaded to clients, resulting in higher throughput for concurrent requests.

Besides, 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 GCC (GNU Compiler Collection) and RDMA atomic primitives. To read or write file data, the locking operation is executed by the server locally using GCC atomic instructions. The unlock operation is executed remotely by the client with RDMA atomic verbs after data I/Os. Note that serializability between GCC and RDMA atomic primitives is not guaranteed due to lack of atomicity between the CPU and the NIC [10, 41, 19]. In Octopus, GCC 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 firstly 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-base RPC has relatively high latency and low throughput. `send/recv` in UD (Unreliable Datagram) mode provides higher throughput [20], but is not suitable for distributed file systems 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 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 queue pair (QP), and thus gets immediately processing after the request arrives. The identifier attached in the immediate field helps the server to direct locate the new message without scanning the whole buffer. After

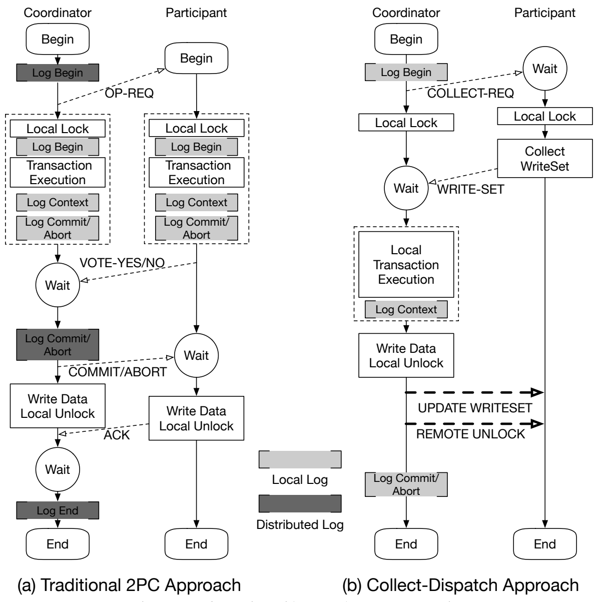

Figure 6: Distributed Transaction

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 than `send/recv` and `read/write` approaches.

3.3.2 Collect-Dispatch Transaction

A single file system operation, like *mkdir*, *mknod*, *rmnod* and *rmdir* in Octopus, performs updates to multiple servers. 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) 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 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 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 GCC 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 GCC `compare_and_swap` command in both coordinator and participants. For the unlock operations, the coordinator releases the local lock using the GCC `compare_and_swap` command 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, (2) RDMA write/atomic primitive does not involve CPU processing of remote side. Thus, we conclude collect-dispatch is efficient, as it not only removes complex negotiations for log persistence ordering across servers, but reduces costly RPC and CPU processing overheads.

Consistency Discussions. In persistent memory systems, data cache in the CPU cache needs to be flushed to the memory timely and ordered to provide crash consistency [11, 26, 33, 25, 14, 32]. In Octopus, metadata consistency is guaranteed by the collect-dispatch transaction, which uses `clflush` to flush data from the CPU cache to the memory to force persistence of the log. While the collect-dispatch transaction can be used to provide data consistency, data I/Os are not wrapped in a transaction in current Octopus implementation for efficiency. We expect that RDMA will have more efficient remote flush operations that could benefit data consistency, such as novel I/O flows like RDMA read for remote durability [12], new proposed commands like `RDMA commit` [39], or new designs that leverage availability for crash consistency [45]. We leave efficient data consistency for future work.

4 Evaluation

In this section, we evaluate Octopus's overall data and metadata performance, then the benefits from each mechanism design, and finally its performance for big data applications.

4.1 Experimental Setup

Evaluation Platform. In the evaluation, we run Octopus on servers with large memory. Each server is equipped with 384GB DRAM and two 2.5GHz Intel Xeon E5-2680 v3 processors, and each processor has 24 cores. Clients run on different servers. Each client server has 16GB DRAM and one Intel Xeon E2620 processor. All these servers are connected with a Mellanox SX1012 switch using CX353A ConnectX-3 FDR HCAs (which support 56 Gbps over InfiniBand and 40GigE). All of them are installed with Fedora 23.

Evaluated File Systems. Table 1 lists the distributed file system (DFSs) for comparison. All these file systems are deployed in memory of the same cluster. For existing DFSs that require local file systems, we build local file systems on DRAM with *pmem* driver and *DAX* [5] supported in *ext4*. The EXT4-DAX [4] is optimized for NVM which bypasses the page cache and reduces memory copies. Octopus manages its storage space on the emulated persistent memory using shared memory (`SHM`) of Linux in each server. These file systems are allocated with 20GB for file system storage at each server. For the network part, all distributed file systems run on RDMA directly. Specifically, memGluster supports using RDMA protocol for communication between glusterfs clients and glusterfs bricks. NVFS is an optimized version of HDFS which exploits the advantages of byte-addressability of NVM and RDMA. Crail is a recent open-source DFS from IBM, and it relies on DaRPC [37] for RDMA optimization and reserves huge pages as transfer cache for bandwidth improvement.

Table 1: Evaluated File Systems

memGluster	GlusterFS runs on memory, and GlusterFS is a widely-used DFS that has no centralized metadata services and is now a part of Redhat
NVFS [16]	a version of HDFS that is optimized with both RDMA and NVM
Crail [9]	an in-memory RDMA-optimized DFS built with DaRPC [37]
memHDFS [35]	HDFS runs on memory, and HDFS is a widely-used DFS for big data processing
Alluxio[23]	an in-memory file system for big data processing

Workloads. In our evaluation, we compare Octopus with memGluster, NVFS and Crail for metadata and read-write performance, and compare it with NVFS and Alluxio for big data benchmarks. 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. For big data evaluation, we replace HDFS by adding Octopus plugin under Hadoop. We use three package-in MapReduce benchmarks in Hadoop, i.e., TestDFSIO, Teragen, and Wordcount, for evaluation.

4.2 Overall Performance

To evaluate Octopus, we first compare its overall performance with memGluster, NVFS and Crail. All these file systems are running in the memory level with RDMA-enabled InfiniBand network. In this evaluation, we first compare Octopus's latency and bandwidth to the raw network's and storage's latency and bandwidth, and then compare Octopus's metadata and data performance to other file systems.

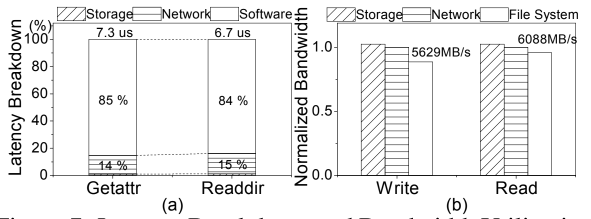

Figure 7: Latency Breakdown and Bandwidth Utilization

4.2.1 Latency and Bandwidth Breakdown

Figure 7 shows both single round-trip latency and bandwidth breakdown for Octopus. From the figures, we have two observations.

(1) The software latency is dramatically reduced to $6us$ (around 85% of the total latency) in Octopus, from $323us$ (over 99%) in memGluster, as shown in Figure 7(a). For the memGluster on the emerging non-volatile memory and RDMA hardwares, the file system layer has a latency that is several orders larger than that of storage or 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. The software latency in Octopus is in the same order with the hardware.

(2) Octopus achieves read/write bandwidth that approaches the raw network bandwidth, as shown in Figure 7(b). The raw storage and network bandwidths respectively are $6509 MB/s$ (with single-thread memcpy) and $6350 MB/s$. Octopus achieves a read/write ($6088/5629 MB/s$) bandwidth that is 95.9%/88.6% of the network bandwidth. In conclusion, Octopus effectively exploits the hardware bandwidth.

4.2.2 Metadata Performance

Figure 8 shows the file systems' performance in terms of metadata IOPS with different metadata operations by varying the number of data servers. From the figure, we make two observations.

(1) Octopus has the highest metadata IOPS among all evaluated file systems in general. memGluster and NVFS provide metadata IOPS in the order of 10^4. Crail provides metadata IOPS in the order of 10^5 owing to DaRPC, a high performance RDMA-based RPC. Comparatively, Octopus provides metadata IOPS in the order of 10^6, which is two orders higher than memGluster and NVFS. Octopus achieves the highest throughput

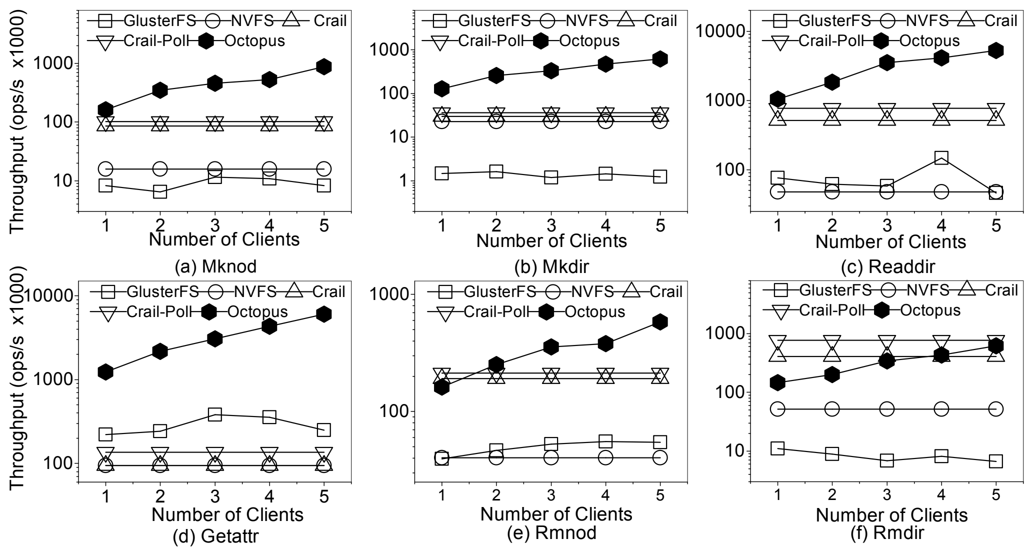
Figure 8: Metadata Throughput

except for *rmdir* and *rmnod* when there is only one data server. Crail is slightly better in this case, because it is deployed with *RdmaDataNode* mode without transaction guarantee. Generally, Octopus achieves high throughput in processing metadata requests, which mainly owes to the self-identified RPC and collect-dispatch transaction that promise extremely low latency and high throughput.

(2) Octopus achieves much better scalability than the other evaluated file systems. NVFS and Crail are designed with single metadata server, and achieve constant metadata throughput. Even with one metadata server, Octopus achieves better throughput than these two file systems in most cases. memGluster achieves the worst throughput, for 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. Besides, memGluster stacks its data management layer on top of the local file system in each server to process metadata requests, and this also limits the throughput. Comparatively, Octopus has the best scalability. For all evaluated metadata operations, Octopus's IOPS is improved by 3.6 to 5.4 times when the number of servers is increased from 1 to 5.

4.2.3 Read/Write Performance

Figure 9 shows the file systems' performance in terms of concurrent read/write throughput with multiple clients by varying the read/write sizes. From figure 9, we can see that, with small read/write sizes, Octopus achieves much higher throughput than other file systems (750 Kops/s and 1 Mops/s for writes and reads respectively). This benefit mainly comes from the client-active data I/O and self-identified RPC mechanisms. NVFS achieves relatively high throughput when read/write size is set to 1KB, for its buffer manager prefetches data to boost

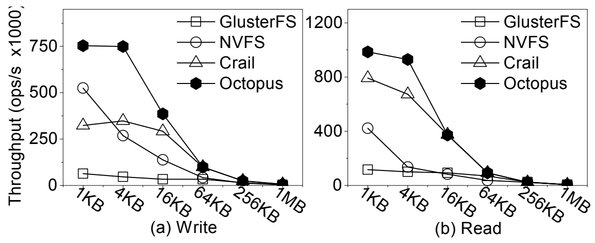
Figure 9: Data I/O Throughput (Multiple Clients)

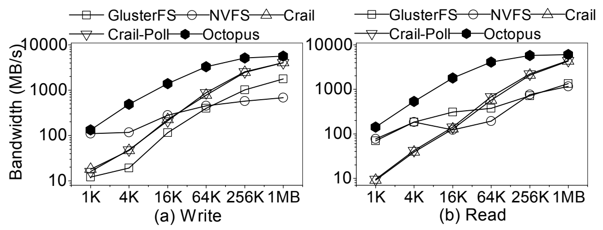
Figure 10: Data I/O Bandwidth (Single Client)

performance. But it drops rapidly when the I/O size grows, which is mainly restricted by the performance of RPC efficiency. Crail has lower throughput than NVFS when I/O size is small, but it achieves throughput close to Octopus when I/O size grows. memGluster has the worst throughput and only achieves 100 Kops/s.

Figure 10 shows the read/write bandwidth achieved by a single client with different read/write sizes. As shown in the figure, Octopus significantly outperforms existing DFSs in terms of read or write bandwidth. When the I/O size is set to 1MB, the read/write bandwidths in NVFS and memGluster are around only $1000MB/s$ and $1500MB/s$, respectively. Crail reaches a bandwidth of $4000MB/s$, which only occupies 63% of the raw network bandwidth. In contrast, Octopus can achieve bandwidth close to that of the raw InfiniBand network ($6088MB/s$ and $5629MB/s$ with 1MB I/O size for read and write respectively), which is mainly because of reduced memory copies by using a shared persistent memory pool.

4.3 Evaluation of Data Mechanisms

4.3.1 Effects of Reducing Data Copies

Octopus improves data transfer bandwidth by reducing memory copies. To verify the effect of reducing data copies, we implement a version of Octopus which add an extra copy at client side, and we refer to it as Octopus+copy. As shown in Figure 11, when I/O size is set to 1MB, Octopus+copy achieves nearly the same bandwidth as Crail (around $4000MB/s$). However, when the extra data copy is removed, Octopus can provide $6000MB/s$ of bandwidth that is written or read by a single client, 23% of extra bandwidth gained. When the I/O size is small, Octopus+copy still surpasses Crail with higher bandwidth, owing to closely coupled RDMA and file system mechanism designs to be evaluated next.

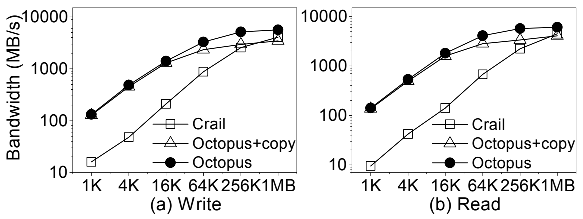

Figure 11: Effects of Reducing Data Copies

4.3.2 Effects of Client-Active Data I/O

We then compare the IOPS of data I/O in client-active and server-active modes that are mentioned in Section 3. Figure 12 shows the read/write throughput of both client-active and server-active modes of Octopus by varying read/write sizes. Crail's performance is also given for reference. We observe that the client-active mode has higher data throughput than the server-active mode for small read/write sizes. Both modes have close throughput for read/write sizes that are larger than 16KB. When the read/write sizes are smaller than 16KB, the client-active mode has higher data throughput by 193% for writes and 27.2% for reads on average. Even the client-active mode consists more network round-trips, it is more efficient to offload workloads to clients from servers when the read/write size is small, in order to improve the data throughput. Client-active mode improves write throughput more obviously than read throughput, because the server side has higher overhead for writes than reads in server-active mode. In server-active mode, after the server side reads data from the client using RDMA `read` when processing client's write operation, it has to check the completion of this operation, which is time-consuming. But for client's read operations, server side never checks the completion message, and provides relatively higher throughput. In all, we conclude that client-active mode has higher bandwidth than the commonly-used server-active mode.

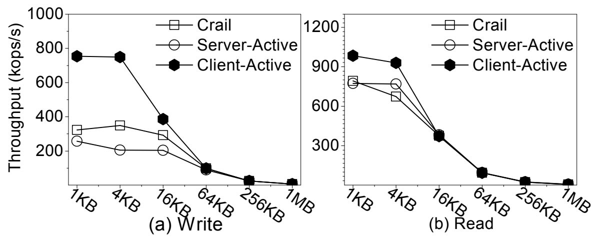

Figure 12: Client-Active Data I/O Performance

4.4 Evaluation of Metadata Mechanisms

4.4.1 Effects of Self-Identified Metadata RPC

We first compare raw RPC performance with different usage of RDMA primitives to evaluate the effects of self-identified metadata RPC. We then compare Octopus with existing file systems on metadata latencies.

Figure 13(a) shows the raw RPC throughput using three RPC implementations (i.e., message-based, memory-based, and self-identified, without message batch) along with DaRPC by varying the I/O sizes. DaRPC used in Crail is designed based on RDMA `send/recv`, and it achieves the lowest throughput, $2.4Mops/s$ with an I/O size of 16 bytes. Its performance may be limited by the Java implementation in its jVerbs interface. We also implement a message-based RPC that uses RDMA `send/recv` verbs, and it achieves a throughput of $3.87Mops/s$ at most. This throughput is limited by the raw performance of RDMA `send/recv`. For the memory-based RPCs that use RDMA `write` verbs, as taken in FaRM [13], we compare the performance by setting the maximum number of client threads to 20 and 100. As observed, the throughput is the highest (i.e., $5.4Mops/s$) when the maximum number of client threads is 20. However, it decreases quickly to $3.46Mops/s$ when the maximum number of client threads is 100. This shows the inefficiency in processing and notification in the memory-based RPCs when there are a large number of client threads. Our proposed self-identified RPC, which carry on client identifiers with the RDMA `write_with_imm` verbs, keeps constant high throughput for an average of $5.4Mops/s$, without being affected by the number of client threads. Similarly, we also measure the latency of each RPC (in Figure 13(b)), among which self-identified RPC keeps relative low latency. As such, self-identified RPCs provide scalable and low-latency accesses, which is suitable for distributed storage systems to support a large number of client requests.

Figure 14 shows metadata latencies of Octopus along with other file systems. As shown in the figure, Octopus achieves the lowest metadata latencies among all the evaluated file systems for all evaluated metadata operations (i.e., $7.3us$ and $6.7us$ respectively for `getattr`

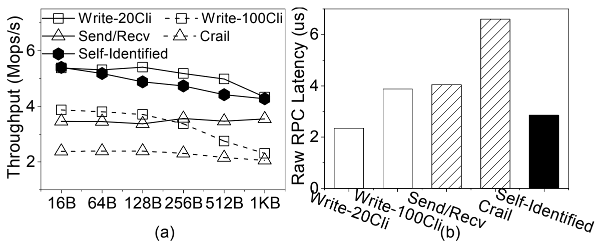

Figure 13: Raw RPC Performance

and `readdir`), which are close to the InfiniBand network latency for most cases. With the self-identified metadata RPC, Octopus can support low-latency metadata operations even without client cache. Crail uses DaRPC for inter-server communication. However, Crail's metadata (e.g., `mkdir` and `mknod`) latencies are much higher than raw DaRPC's latency. This possibly is because Crail is implemented on the inefficient HDFS framework, or it registers memory temporarily for message communication, which is time-consuming. NVFS and memGluster suffer the similar problem of heavy file system designs as Crail, and thus have relatively higher latency.

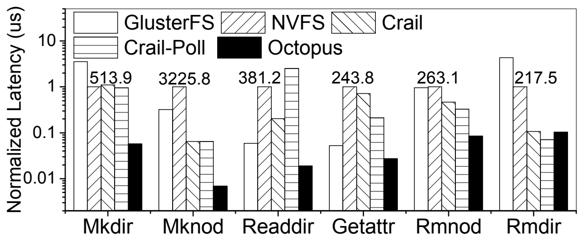

Figure 14: Metadata Latency

4.4.2 Effects of Collect-Dispatch Transaction

To evaluate the effects of the collect-dispatch transaction in Octopus, we also implement a transaction system based on 2PC for comparison. Figure 15(a) exhibits the latencies of these two transaction mechanisms. Collect-dispatch reduces latency by up to 37%. This is because 2PC involves two RPCs to exchange messages from remote servers, while collect-dispatch only needs one RPC and two one-sided RDMA commands to finish the transaction. Although the number of messages is increased, the total latency drops. RPC protocol needs the involvements of both local and remote nodes, and a lot of side information (e.g., hash computing, and message discovery) needs to be processed at this time. Thus, RPC latency (around $5us$) is much higher than one-sided RDMA primitives (less than $1us$). From figure 15(b) we can see that, transaction based on collect-dispatch improves throughput by up to 79%. On one hand, collect-dispatch only writes logs locally, significantly reducing logging overhead. On the other hand, collect-dispatch decreases the total number of RPC when processing transactions, which reduces the involvements of remote CPUs and thereby improves performance.

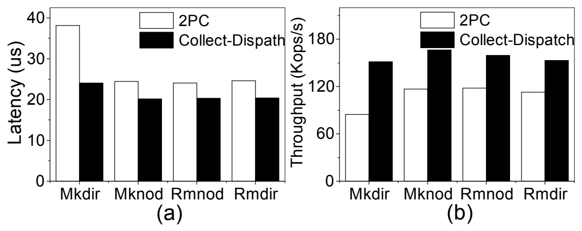

Figure 15: Collect-Dispatch Transaction Performance

4.5 Evaluation using Big Data Applications

In addition, we compare Octopus with distributed file systems that are used in big data framework. We configure Hadoop with different distributed file systems - memHDFS, Alluxio, NVFS, Crail and Octopus. In this section, we compare both read/write bandwidth and application performance.

Read/Write Bandwidth. Figure 16(a) compares the read/write bandwidths of above-mentioned file systems using *TestDFSIO* by setting the read/write size to 256KB. Octopus and Crail show much higher bandwidth than traditional file systems. Octopus achieves $2689MB/s$ and $2499MB/s$ for write and read operations respectively, and Crail achieves $2424MB/s$ and $2215MB/s$ respectively. Note that they have lower bandwidths than the results in `fio`. The reason is that we connect Octopus/Crail with Hadoop plugin using JNI (Java Native Interface), which restricts the bandwidth. In contrast, memHDFS, Alluxio and NVFS show lower bandwidth than Octopus and Crail. memHDFS has the lowest bandwidth, for the heavy HDFS software design that is for hard disks and traditional Ethernet. Alluxio and NVFS are optimized to run on DRAM, and thus provide higher bandwidth than memHDFS. But they are still slower than Octopus. Thus, we conclude the general-purpose Octopus can also be integrated into existing big data framework and provide better performance than existing file systems.

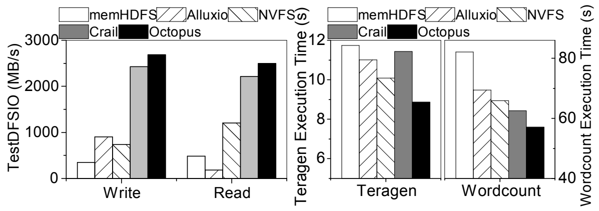

Figure 16: Big Data Evaluation

Big Data Application Performance. Figure 16(b) shows the application performance for different file systems. Octopus consumes the least time to finish all evaluated applications. Among all the evaluated file systems, memHDFS generally has the highest run time, i.e., $11.7s$ for Teragen and $82s$ for Wordcount. For the Teragen workload, the run time in Alluxio, NVFS, Crail and Octopus is $11.0s$, $10.0s$, $11.4s$ and $8.8s$, respectively.

For the Wordcount workload, the run time in Alluxio, NVFS, Crail and Octopus is $69.5s$, $65.9s$, $62.5s$ and $57.1s$, respectively. We conclude that our proposed general-purpose Octopus can even provide better performance for big data applications than existing dedicated file systems.

5 Related Work

Persistent Memory File Systems: In addition to file systems that are built for flash memory [17, 28, 27, 22, 44], a number of local file systems have been built from scratch to exploit both byte-addressability and persistence benefits of non-volatile memory [11, 14, 42, 32, 43]. BPFS [11] is a file system for persistent memory that directly manages non-volatile memory in a tree structure, and provides atomic data persistence using short-circuit shadow paging. PMFS [14] proposed by Intel also enables direct persistent memory access from applications by removing file system page cache with memory mapped IO. Similar to BPFS and PMFS, SCMFS [42] is a file system for persistent memory which leverages the virtual memory management of the operating system. Fine-grained management is further studied in recent NOVA [43] and HiNFS [32] to make software more efficient. The Linux kernel community also starts to support persistent memory by introducing DAX (Direct Access) to existing file systems, e.g., EXT4-DAX [4]. The efficient software design concept in these local file systems, including removing duplicated memory copies, is further studied in Octopus distributed file system to make remote accesses more efficient.

General RDMA Optimizations: RDMA provides high performance but requires careful tuning. Recent study [19] offers guidelines on how to use RDMA verbs efficiently from a low-level perspective such as in PCIe and NIC. Cell [30] dynamically balances CPU consumption and network overhead using RDMA primitives in a distributed B-tree store. PASTE [15] proposes direct NIC DMA to persistent memory to avoid data copies, for a joint optimization between network and data stores. FaSST [20] proposes to use UD (Unreliable Datagram) for RPC implementation when using `send/recv`, in order to improve scalability. RDMA has also been used to optimize distributed protocols, like shared memory access [13], replication [45], in-memory transaction [41], and lock mechanism [31]. 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 [29, 18, 13, 40]. MICA [24] bypasses the kernel and uses a lightweight networking stack to improve data access performance in key-value stores. Pilaf [29] optimizes the *get* operation using multiple RDMA `read` commands at the client side, which offloads hash calculation burden from remote servers to clients, improving system performance. HERD [18] implements both *get* and *put* operations using the combination of RDMA `write` and UD `send`, in order to achieve high throughput. HydraDB [40] is a versatile *key-value* middleware that achieves data replication to guarantee fault-tolerance and awareness for NUMA architecture, and adds client-side cache to accelerate the *get* operation. While RDMA techniques lead to evolutions in the designs of key-value stores, its impact on file system designs is still under-exploited.

RDMA Optimizations in Distributed File Systems: Existing distributed file systems have tried to support RDMA network by substituting their communication modules [1, 3, 6]. Ceph over Accelio [3] is a project under development to support RDMA in Ceph. Accelio [2] is an RDMA-based asynchronous messaging and RPC middleware designed to improve message performance and CPU parallelism. Alluxio [23] in Spark (formerly named Tachyon) is transplanted to run on top of RDMA by Mellanox [6]. It faces the same problem as Ceph on RDMA. NVFS [16] 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 [9] is a recently developed distributed file system built on DaRPC [37]. 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.

6 Conclusion

The efficiency of the file system design becomes an important design issue for storage systems that are equipped with high-speed NVM and RDMA hardware. Both the two emerging hardware technologies not only improve hardware performance, but also push back the software evolution. In this paper, 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 show that Octopus effectively explores hardware benefits, and significantly outperforms existing distributed file systems.

Acknowledgments

We thank our shepherd Michio Honda and anonymous reviewers for their feedbacks and suggestions. We also thank Weijian Xu for his contribution in the early prototype of Octopus. This work is supported by the National Natural Science Foundation of China (Grant No. 61502266, 61433008, 61232003), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), and the China Postdoctoral Science Foundation (Grant No. 2016T90094, 2015M580098). Youyou Lu is also supported by the Young Elite Scientists Sponsorship Program of China Association for Science and Technology (CAST).

References

[1] GlusterFS on RDMA. "https://gluster.readthedocs.io/en/latest/AdministratorGuide/RDMATransport/".

[2] Accelio. "http://www.accelio.org", 2013.

[3] Ceph over Accelio. "https://www.cohortfs.com/ceph-over-accelio", 2014.

[4] Support ext4 on NV-DIMMs. "https://lwn.net/Articles/588218", 2014.

[5] Supporting filesystems in persistent memory. "https://lwn.net/Articles/610174", 2014.

[6] Alluxio on RDMA. "https://community.mellanox.com/docs/DOC-2128", 2015.

[7] Introducing Intel Optane technology - bringing 3D XPoint memory to storage and memory products. "https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/", 2016.

[8] SAP HANA, in-memory computing and real time analytics. "http://go.sap.com/product/technology-platform/hana.html", 2016.

[9] Crail: A Fast Multi-tiered Distributed Direct Access File System. https://github.com/zrlio/crail, 2017.

[10] Association, I. T., et al. *InfiniBand Architecture Specification: Release 1.3*. InfiniBand Trade Association, 2009.

[11] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (New York, NY, USA, 2009), ACM, pp. 133–146.

[12] Douglas, C. RDMA with PMEM: software mechanisms for enabling access to remote persistent memory. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf, 2015.

[13] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 401–414.

[14] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (New York, NY, USA, 2014), ACM, pp. 15:1–15:15.

[15] Honda, M., Eggert, L., and Santry, D. Paste: Network stacks must integrate with nvmm abstractions. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), ACM, pp. 183–189.

[16] Islam, N. S., Wasi-ur Rahman, M., Lu, X., and Panda, D. K. High performance design for hdfs with byte-addressability of nvm and rdma. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ACM, p. 8.

[17] Josephson, W. K., Bongo, L. A., Flynn, D., and Li, K. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)* (Berkeley, CA, 2010), USENIX.

[18] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *SIGCOMM* (2014).

[19] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[20] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association, pp. 185–201.

[21] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 2009), ACM, pp. 2–13.

[22] Lee, C., Sim, D., Hwang, J., and Cho, S. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, Feb. 2015), USENIX.

[23] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (2014).

[24] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. *management 15*, 32 (2014), 36.

[25] Lu, Y., Shu, J., and Sun, L. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st Conference on Massive Storage Systems and Technologies (MSST)* (2015), IEEE, pp. 1–13.

[26] Lu, Y., Shu, J., Sun, L., and Mutlu, O. Loose-ordering consistency for persistent memory. In *Proceedings of the IEEE 32nd International Conference on Computer Design (ICCD)* (2014), IEEE.

[27] Lu, Y., Shu, J., and Wang, W. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)* (Berkeley, CA, 2014), USENIX, pp. 75–88.

[28] Lu, Y., Shu, J., and Zheng, W. 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)* (Berkeley, CA, 2013), USENIX.

[29] Mitchell, C., Geng, Y., and Li, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 103–114.

[30] Mitchell, C., Montgomery, K., Nelson, L., Sen, S., and Li, J. Balancing cpu and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[31] NARRAVULA, S., MARNIDALA, A., VISHNU, A., VAIDYANATHAN, K., AND PANDA, D. K. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)* (2007), IEEE, pp. 583–590.

[32] OU, J., SHU, J., AND LU, Y. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 12.

[33] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)* (2014), pp. 265–276.

[34] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 2009), ACM, pp. 24–33.

[35] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *IEEE 26th symposium on mass storage systems and technologies (MSST)* (2010), IEEE, pp. 1–10.

[36] STRUKOV, D. B., SNIDER, G. S., STEWART, D. R., AND WILLIAMS, R. S. The missing memristor found. *nature 453*, 7191 (2008), 80–83.

[37] STUEDI, P., TRIVEDI, A., METZLER, B., AND PFEFFERLE, J. DaRPC: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2014), ACM, pp. 1–13.

[38] SWANSON, S., AND CAULFIELD, A. M. Refactor, reduce, recycle: Restructuring the i/o stack for the future of storage. *Computer 46*, 8 (2013), 52–59.

[39] TALPEY, T. Remote Access to ultra-low-latency storage. `http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/Talpey-Remote_Access_Storage.pdf`, 2015.

[40] WANG, Y., ZHANG, L., TAN, J., LI, M., GAO, Y., GUERIN, X., MENG, X., AND MENG, S. 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* (2015), ACM, p. 22.

[41] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 87–104.

[42] WU, X., AND REDDY, A. L. N. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (New York, NY, USA, 2011), ACM, pp. 39:1–39:11.

[43] XU, J., AND SWANSON, S. Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 323–338.

[44] ZHANG, J., SHU, J., AND LU, Y. Parafs: A log-structured file system to exploit the internal parallelism of flash devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[45] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 3–18.

[46] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 2009), ACM, pp. 14–23.