NVMe-Driven Lazy Cache Coherence for Immutable Data with NVMe over Fabrics

Tuan Anh Nguyen*, Hyeongjun Jeon*, Daegyu Han*¶ , Duck-Ho Bae†, Young Jin Yu†,
Kyeungpyo Kim‡, Sungsoon Park‡, Jinkyu Jeong§, Beomseok Nam*
*Sungkyunkwan University, †Samsung Electronics, ‡GlueSys, §Yonsei University

***Abstract*—In this work, we explore opportunities to design shared storage systems that leverage the distance connectivity of NVMe over Fabrics (NVMe-oF). NVMe-oF enables the use of NVMe storage devices in a shared storage environment, where multiple servers can access the same storage device via RDMA. Leveraging the distance connectivity of NVMe-oF, we develop a shared file system called *EXT4-oF* by extending the EXT4 file system. EXT4-oF uses RDMA to enable a local file system to function as a shared file system without requiring remote daemon processes. EXT4-oF employs a novel *NVMe-driven lazy cache coherence* to maintain cache coherence of file system metadata across multiple compute nodes upon creating new files, all achieved without the need for any daemon processes. To ensure cache coherence, NVMe-driven lazy cache coherence mechanism requires compute nodes to perform a re-read of NVMe-oF to avoid false negative file open errors. Through our experiments, we demonstrate that EXT4-oF improves the performance of MinIO by minimizing network traffic between compute and storage nodes and eliminating the need for TCP/IP communication during remote reads.**

I. Introduction

NVMe-oF (NVMe over Fabrics) is a network protocol specification designed to allow NVMe commands to be transported over a network fabric instead of a local PCIe bus [1]. Through the use of RDMA, NVMe-oF provides disaggregated access to a remote NVMe device as if it were local, even though the device is being shared by multiple servers. Without NVMe-oF, NVMe devices are limited to being local to a single host [15].

Recently, NVMe-oF has gained significant attention, particularly in the industry community, due to its similar objectives with Software-Defined-Storage (SDS). Both technologies decouple physical storage devices from compute nodes and offer scalable logical storage services via on-demand provisioning of storage capacity[14]. However, SDS suffers from a thick software layer incurring millisecond-order latencies. In contrast, NVMe-oF adds less than 10 microseconds of latency as it eliminates protocol translation overhead and bypasses storage host's CPU and software stack[8].

In comparison to SDS, NVMe-oF has certain limitations that make it less flexible for shared storage systems[4]. Specifically, NVMe-oF does not guarantee consistency for concurrent accesses to the same NVMe device, as it exposes a raw device instead of a shared file system interface to compute nodes. For instance, if a Linux native file system of an NVMe partition is mounted on multiple nodes, file systems on different nodes may attempt to write to the same block simultaneously, leading to inconsistencies.

Consequently, NVMe-oF is currently being used as a supplementary method of disaggregating storage systems rather than directly competing with SDS. For example, Ceph provides an option to deploy a storage node with NVMe-oF devices as the second tier behind OSD (Object Storage Daemon) nodes, adding another storage tier in the already multi-tiered SDS [3]. This approach hides one of the most promising features of NVMe-oF, which is the direct connectivity of NVMe-oF.

In this work, we explore the untapped potential of NVMe-oF, which enables multiple computing nodes to share data without the need for a shared file system. In particular, we propose to make small modifications to existing Linux native file systems to develop shared file systems for immutable data. We modify the EXT4 file system to develop a shared file system called *EXT4-oF*, which enables multiple nodes to share files via NVMe-oF without the need for any background daemon process running on a storage node. By minimizing the number of network hops and reducing overhead on the I/O stack, EXT4-oF ensures that read and write operations to remote files occur at speeds comparable to those to local files.

The NVMe-oF-based shared file system proposed in this study may have file consistency issues when multiple nodes update a file simultaneously, as it does not use daemon processes. However, large amounts of data stored and managed in modern data centers are often characterized by immutability. Immutability has traditionally been a key concept in a variety of computing domains, including functional programming languages, log-structured file systems, copy-on-writes (CoW), DBMS snapshots, and more, although the lifetime of such legacy immutable data is typically very short in those systems.

With the advent of big data processing, machine learning, and blockchain technologies, applications that query *write-once-read-many (WORM)* datasets have become increasingly common in modern data centers [10]. The lifespan of immutable data is now long enough to explore new opportunities in data processing frameworks. Storage systems like HDFS [18], object storage systems such as MinIO, and log-structured merge (LSM) tree-based key-value stores such as HBase, Cassandra, LevelDB, and RocksDB, operate under constraint that once a file is created, it can only be appended and deleted, but not updated. This immutability allows shared file systems to avoid the use of traditional cache coherence

¶Department of Electrical and Computer Engineering

protocols to resolve file consistency issues caused by conflicts between different nodes during the writing and reading process, unlike other distributed file systems.

In this study, we develop a shared file system for managing immutable data by **(1)** *granting write permission for a specific file to only one compute node among multiple nodes in the cluster while allowing other nodes to read the file*, and **(2)** *maintaining metadata consistency by reading file metadata directly from NVMe-oF in a bottom-up manner, without relying on the inode and dentry metadata cached in the existing Linux native file system*.

The contributions of this work are summarized as follows.

- By taking advantage of NVMe-oF's ability to provide distance connectivity and enable sharing of an NVMe between multiple compute nodes, we extend a Linux native file system - EXT4 to behave as a shared file system for immutable data without running network daemon processes and using remote CPUs.
- In order to ensure that metadata for immutable files remains consistent across multiple compute nodes, we design and implement *NVMe-driven lazy cache coherence* mechanism. This mechanism enables files created by one compute node to be visible to other compute nodes, but in a lazy and efficient manner.
- EXT4-of can be used in conjunction with middleware designed for managing immutable data. As a case study, we extended MinIO to incorporate the NVMe-driven lazy cache coherence mechanism of EXT4-oF, resulting in improved performance for remote read through the transformation of remote reads to local reads leveraging the distance connectivity of NVMe-of.
- Our performance study demonstrates that replacing remote file read over a TCP/IP network with local read over NVMe-oF results in improved read throughput and a 50% reduction in CPU utilization. When deployed with MinIO, EXT4-oF improves the performance of read-intensive workloads by up to 1.4×.

II. Background and Motivation

A. NVMe over Fabrics

NVMe-oF is a storage protocol that allows NVMe devices to be accessed over an RDMA-capable network [1, 2, 8]. NVMe-oF has similarities to iSCSI storage in that both protocols send native device commands over the network fabric. However, NVMe-oF is much faster than iSCSI because NVMe-oF uses a low-latency RDMA protocol (e.g., iWARP and Infiniband) to communicate between the host and storage, while iSCSI uses TCP/IP over Ethernet. Besides, NVMe-oF offloads most of the processing to the storage device, reducing the CPU utilization, while iSCSI requires CPU processing on the storage node.

NVMe disks communicate with the CPU via a PCIe connection. Therefore, the OS must be aware of data transfer operations between the local NVMe and the remote host. However, this challenge can be overcome with RDMA because it offloads memory copy operations to the network adapter such that data can be transferred without going through the OS-level network stack. That is, RDMA eliminates the necessity of copying data between application memory and the data buffer in the OS [1]. As a result, NVMe-oF allows applications to directly access a remote NVMe over the network without having to involve the remote storage node's OS or network daemon process.

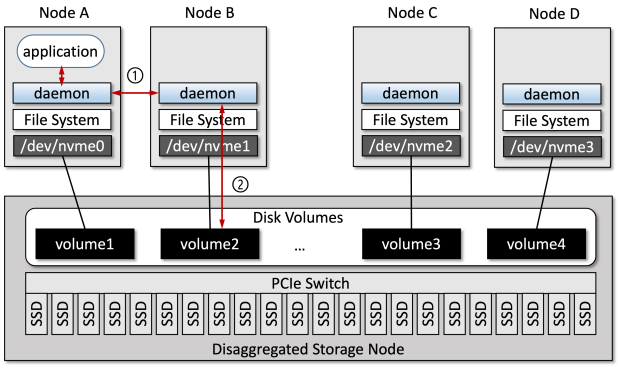


Fig. 1. Remote Read through Middleware

B. Immutability: Write-Once Read-Many Semantics

The recent advent of blockchain, machine learning technologies, and object storage systems is fuelling interest in efficient management of immutable data [10]. Immutable data is trending in such an increasing number of computing domains [10] because it simplifies many things over mutable data. Most importantly, immutability enables idempotent operations, i.e., repeating the same operations on immutable data guarantees the same output. This is an important characteristic in large scale data processing systems where failures are common. For example, distributed file systems such as HDFS and object storage systems such as MinIO exploit immutability for high availability. Specifically, they pipeline replication operations across data centers assuming data cannot be tampered during replication (i.e., *write-once-read-many* (WORM) semantics). Such update prevention greatly simplifies the data consistency model and reduces the bottleneck of the central metadata management service; e.g., NameNode in HDFS does not manage write conflicts on the same file.

C. Remote Reads in Distributed Systems

Distributed storage systems provide transparent access to remote data. However, remote read is one of the main performance bottlenecks in large scale distributed systems. This is because remote read not only incurs high network latency, but also consumes CPU cycles on the remote storage node. As storage devices become faster (e.g., ultra low latency NVMe SSDs) and network bandwidth increases (e.g., 400G Ethernet), CPU is emerging as a new bottleneck in distributed storage systems [11].

In conventional general-purpose POSIX shared file systems such as OCFS2 [7] and GFS2 [19], each node runs a background file system daemon process to provide local POSIX

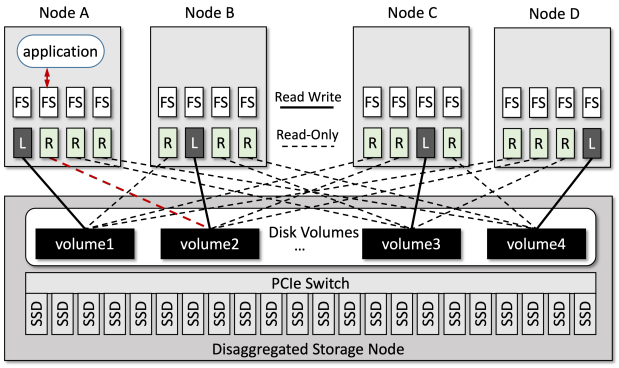


Fig. 2. Remote Read through Local File System and NVMe-oF

file system semantics to applications, such as sequential consistency, location transparency, naming transparency, and more [13]. The file systems daemon processes communicate with each other to guarantee the coherence of distributed caches and to control concurrent accesses to a shared file for file system consistency.

Figure 1 shows an architecture of a conventional user-level distributed file system such as HDFS deployed on a cluster with a disaggregated storage box. Each NVMe-oF disk volume of the storage box is dedicated to a compute node as if it is a local direct attached storage (DAS). Suppose an application running on Node A reads a file stored in Node B's NVMe volume. The application interacts with the local daemon process, which forwards the read request to the remote daemon process running on Node B (①) through TCP/IP network. Then, the Node B's daemon process reads the file from its local NVMe-oF volume (`/dev/nvme1`). Since the NVMe-oF volume is in a disaggregated storage box, the read request goes through RDMA network (②). Once the file data is copied to the memory buffer that Node B's daemon process manages, it is sent back to Node A's daemon process through another TCP/IP network connection.

We note that the remote data access requires to go through TCP/IP stack twice. Therefore, it not only consumes a non-negligible number of CPU cycles in both nodes, but also increases the network traffic. Another drawback of this conventional configuration is that we may lose access to an NVMe if a remote *intermediator* node (Node B in the example) fails. Although there exist various mechanisms that improve the fault tolerance, e.g., replication and erasure coding, the level of fault tolerance can be improved if the intermediator node is eliminated from the IO path in the cluster.

Considering NVMe-oF provides direct distance connectivity to remote NVMe devices over network transports such as RDMA and TCP, we aim to eliminate the intermediate node and reduce the number of network hops to benefit from low disk access latency comparable to DAS [5, 16, 20, 9], i.e., we let applications access remote files directly via NVMe-oF.

III. Design of EXT4-oF

In this section, we present an experimental shared file system - EXT4-oF, which eliminates the need for intermediator daemon processes using NVMe-oF, and thereby reduces the number of network hops, network traffic, and the use of CPU cycles across the cluster.

Taking advantage of the immutability of modern datasets, EXT4-oF simplifies the consistency guarantee mechanism of shared network storage systems. In return for that benefit, EXT4-oF does not provide strong POSIX semantics for general-purpose applications, but only supports WORM (write-once-read-many) semantics, where file-backed pages are naturally guaranteed to be cache-coherent.

EXT4-oF is a variant of Linux native EXT4 file system; i.e., applications open, read, write, unlink, and close a file using POSIX file system interfaces. From the application's point of view, EXT4-oF is no different from EXT4, and the physical file system layout and metadata managed by EXT4-oF are also identical to those of EXT4. However, EXT4-oF is not a file system for a single node, but a file system that allows direct reading of files managed by remote nodes, as in a shared file system.

In EXT4-oF, to enable remote reads through a local filesystem - EXT4, a dedicated NVMe volume on the remote node is also mounted using a local filesystem, as shown in Figure 2. With this *all-to-all* configuration, we mount each NVMe-oF SSD in its own mount point on every node. This allows applications to directly access any file on any NVMe SSD without going through the TCP/IP stack, using POSIX system calls such as `open()`, `read()`, `write()`, and `close()`. For example, the application on Node A can access NVMe `volume2` without communicating with a daemon process running on Node B in Figure 1).

However, when using a file system for a single node to allow multiple nodes to share files, there may be issues with file system consistency. That is, while traditional shared file systems limit the number of nodes that can perform read and write operations on a specific NVMe volume to one, using a non-shared file system with a single NVMe-oF mounted on multiple nodes for access can result in conflicts if multiple nodes attempt to update the same file. Even with immutable data, multiple nodes may compete to write to the same disk block when creating their files or objects. Shared file systems implement mechanisms to resolve such conflicts, but non-shared Linux file systems like EXT4, which lack daemon processes, cannot resolve such conflicts. Moreover, if multiple nodes mount the same NVMe-oF SSD using a non-shared Linux file system such as EXT4, each node will have its own file system metadata and it may cause cache coherence problems regarding cached file system metadata.

In the following, we describe how EXT4-oF resolves such file system consistency issues.

A. Single Writer Node

In conventional shared file systems, daemon processes communicate with each other via TCP/IP to control concurrent

access to a shared file. Specifically, daemon processes grant write permission to a single client while blocking all other read or write requests. However, EXT4-oF does not employ daemon processes, and communication between nodes is only possible through NVMe. Therefore, resolving conflicts between concurrent accesses in EXT4-oF requires a different approach.

EXT4-oF is specifically tailored for immutable data. When files are immutable, complex protocols and costly file system daemon processes to resolve write-write conflicts become unnecessary. Nonetheless, when multiple writers operate on different nodes, they may unintentionally assign the same free disk block for each file. This can happen because each node independently manages the file system metadata, including free blocks. Such a write-write conflict can lead to corruption of the file system.

To prevent such consistency issues without relying on TCP/IP communications, EXT4-oF grants write permission for each NVMe-oF disk to a single node. This *single writer node policy* ensures that only one node can write to a disk and avoid write conflicts, while all other nodes have read-only access. The grant of write access to a particular node for each disk is permanent unless the system administrators modify the configuration. While it may initially seem like a significant limitation, having only one node with write access is not a unique restriction of EXT4-oF. Traditional distributed file systems also limit the number of nodes with read/write access to one since disks have historically been a resource that wasn't shared across nodes until recently. As a result, the concurrency level of EXT4-oF is not lower than that of legacy shared file systems.

Under the single writer node policy, each node must be able to differentiate between writable and read-only NVMe disks. EXT4-oF allows applications to determine the mapping between nodes and NVMe disks. We refer to a writable file system as a *pseudo-local* file system and a read-only file system as a *pseudo-remote* file system. In Figure 2, pseudo-local file systems with read and write permission are denoted as `L`, and pseudo-remote file systems with read-only permission are denoted as `R`. Specifically, Node A's pseudo-local file system is `volume1`, which is exclusive to Node A with read/write permission, while the other nodes have only read-only access to it.

B. Metadata Cache Coherence

The single writer node policy helps resolve conflicts that occur when managing the free block list, but non-shared Linux native file systems such as EXT4 manage the file system's metadata in a local cache and do not share it with other nodes. This means that file system metadata updated in a pseudo-local file system that shares an NVMe-oF is not reflected in the pseudo-remote file system.

For instance, suppose there is an NVMe-oF mounted on both Node A and Node B, as shown in Figure 3. Node B utilizes its pseudo-remote file system to read the file `/dir/a.txt`. In this case, the corresponding *inode*, its *dentry*, and data blocks of `/dir` are cached in local inode and

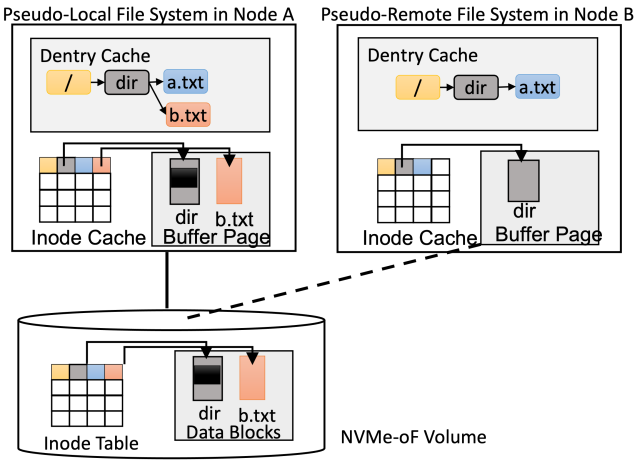


Fig. 3. Metadata Cache Inconsistency

dentry caches. Then, Node A creates a new file named `b.txt` in `/dir/` directory. Although the inode table and data blocks of `/dir` are updated in Node 2's file system and its file system metadata caches are now up to date, Node B's cache remains outdated. When an application on Node B tries to open the `b.txt` file, the metadata for file `b.txt` does not exist in the dentry cache and inode cache of Node B. Hence, the `open()` system call returns with failure. In conventional shared file systems, daemon processes are responsible for updating or invalidating remote caches using TCP/IP communication.

To resolve this challenge, EXT4-oF introduces a new file open flag - `O_REFRESH_DIR_CACHE`. In fact, the only modification made to EXT4 file system in the development of EXT4-oF is the addition of this flag. When the `O_REFRESH_DIR_CACHE` flag is set, EXT4-oF does not trust its locally cached metadata if a file is not found. In other words, if the `open()` system call is about to return with failure, EXT4-oF invalidates the corresponding dentry and inode cache entries, and reads disk blocks to retrieve the latest metadata. This results in the cache being invalidated and updated in a bottom-up manner. We refer to this cache coherence mechanism as the *NVMe-driven lazy cache coherence* protocol.

Let us walk through the same example and describe how the cache coherence problem is resolved using NVMe-driven lazy cache coherence protocol. If an application in Node B attempts to open `b.txt`, EXT4-oF reads the data block for `/dir` from the file system cache. Although the cached data block is outdated and `b.txt` cannot be found, the `open()` does not return with failure and instead invalidates the cached data block for `/dir` while fetching the data block from NVMe again if the `O_REFRESH_DIR_CACHE` flag is set. Additionally, since the inode cache for `/dir` has not been updated, EXT4-oF has to fetch the inode from NVMe and store it in the inode cache. It is noteworthy that the invalidation order is opposite to the normal read flow. Finally, with the revalidated data blocks of `/dir`, EXT4-oF can locate the

inode of `b.txt`, create its dentry, and return a file descriptor.

It is worth noting that the lazy cache coherence mechanism may cause non-negligible overhead if applications frequently create or delete a large number of files in a single directory. However, WORM applications typically create files much less frequently than they read them. Moreover, the files that WORM applications access are usually very large (e.g., 128 MB in HDFS), meaning that the overhead of reading the file metadata from NVMe is significantly less than the benefit of bypassing the remote intermediator daemon process.

IV. EXT4-oF in Action

The primary goal of EXT4-oF, which extends EXT4 to support bottom-up lazy cache coherence, is to improve the performance of remote reads in distributed data processing frameworks by converting them into local reads. In other words, EXT4-oF cannot be used as a shared file system on its own because it does not provide a metadata directory service like conventional distributed file systems do.

A. External Directory Service

A directory service is necessary in distributed file systems to locate files over the network and provide a consistent view of the file system. For example, HDFS cluster consists of a *NameNode* that manages the file system metadata and *DataNodes* that store the actual files [17, 12].

This directory service maintains the hierarchical tree structure of all distributed files and directories, keeps track of where files are created, copied, moved, and deleted, and ensures the consistency of file systems by controlling concurrent accesses to shared files. However, since the directory service requires network communication, it is often a single point of failure.

To make distributed data processing frameworks scale in a fault-tolerant manner, some distributed systems such as Dynamo, Cassandra, and MinIO use consistent hashing to locate data objects across thousands of servers, instead of relying on a centralized directory service.

Regardless of whether distributed systems use a centralized directory service or decentralized consistent hashing, EXT4-oF plays a role of converting remote reads into local reads for these frameworks. That is, EXT4-oF is designed for use with other distributed storage systems. One of the core design principles of EXT4-oF is to minimize network communication, and introducing another distributed directory service is not desirable as it would create another potential point of failure. Instead, EXT4-oF relies on an external directory service in the upper layer distributed systems, such as NameNode in HDFS and consistent hashing in Cassandra and MinIO. This means that it is the responsibility of an upper layer distributed framework to manage the list of shared files and their paths. For example, if a file is created by an HDFS application, the file will be registered in the NameNode. The other DataNodes can then locate the new file by looking up the directory service [6]. The underlying EXT4-oF does not need to know what files are stored in which DataNode [9].

B. MinIO-oF: MinIO on top of EXT4-oF

MinIO is a highly scalable distributed object storage engine that is compatible with the Amazon S3 cloud storage service. It provides a simple yet robust solution for storing and retrieving objects (files) using HTTP requests. One of its notable features of MinIO is that it partitions each new object into data and parity shards through erasure coding. Additionally, it leverages consistent hashing to effectively locate and distribute these shards across multiple nodes. The consistent hashing in MinIO is responsible for managing the file names of shards and ensuring that the underlying Linux file systems always receive valid file I/O requests. Any attempt to access an invalid object will result in a failure at the MinIO level, rather than in the Linux file system.

Remote read in MinIO occurs due to erasure coding. When a client reads an object, MinIO nodes need to retrieve and reconstruct the shards of the partitioned object from an *erasure set* in remote MinIO nodes. The erasure coding improves read performance through IO parallelism, but it generates frequent remote reads.

To utilize the distance connectivity of NVMe-oF, we made small modifications to MinIO, which we refer to as *MinIO-oF*. In MinIO-oF, every node mounts all NVMe volumes and transforms remote reads into local reads through pseudo-remote EXT4-oF file system.

V. Performance Evaluation

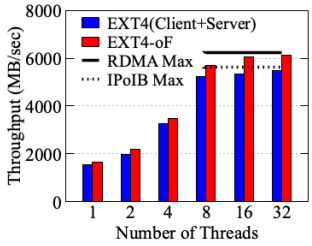
A. Experimental Setup

We conduct a performance evaluation to assess the impact of EXT4-oF on MinIO using a 9-node cluster, i.e., 8 MinIO nodes and 1 disaggregated storage node. Each node was equipped with two 10-core Xeon Gold 5115 processors (2.4GHz/14MB) with hyper-threading enabled and 64GB DRAM. The storage node has two 8-core Xeon Silver 4215 processors (2.50GHz/11MB), 64GB DRAM, and 8 Samsung PM983 U.2 NVMe SSDs (up to 3 GB/s for sequential read and 540,000 kIOPS for random read). All the nodes are connected via a 56 Gbps Mellanox ConnectX-4 Infiniband interconnect.

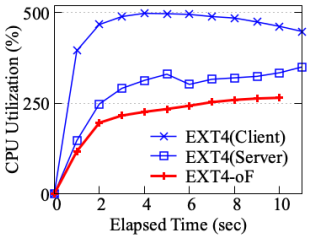
We implemented EXT4-oF in Linux kernel 4.20 (Ubuntu 18.04), and MinIO-oF in MinIO (RELEASE.2022-04-30T22-23-53Z). All the numbers presented in this section represent averages of at least five runs.

B. Performance Evaluation of EXT4-oF

1) Performance Effect of Network Hops: To quantify the overhead of accessing data using multiple network hops as traditional distributed systems do, we modified EXT4 to read remote files using TCP/IP network communication. That is, we implemented a conventional but overly simplified shared file system for immutable data on top of EXT4, which we refer to as *EXT4(Client-Server)*. It is worth noting that EXT4(Client-Server) does not consider distributed cache coherence and shared file system consistency. Therefore, of the various overheads of a shared file system, the only overhead that



(a) Sequential Read



(b) CPU Utilization with 32 Threads

Fig. 4. Remote Read Throughput (EXT4-oF v.s. Client-Server)

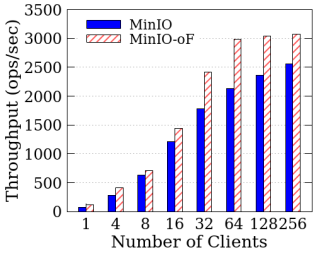


Fig. 5. Read Throughput (YCSB C)

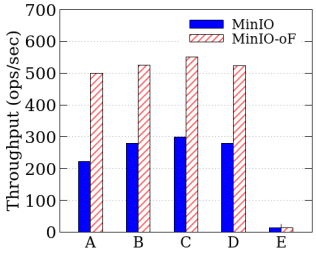


Fig. 6. YSCB (Uniform)

EXT4(Client-Server) has is the TCP/IP network communication overhead.

The experiments presented in Figure 4(a) compares the performance of EXT4-oF and EXT4(Client-Server) by running micro-benchmarks where each thread reads a 2-GB file. To conduct the experiments, we create a RAID 0 volume using three NVMe SSDs on the storage node. We mount the volume using the NVMe-oF protocol for EXT4-oF in both nodes, as shown in Figure 2. For EXT4(Client-Server), we mount the volume in one node in the traditional way, such that the other node accesses the volume through a remote process and TCP/IP communication, as shown in Figure 1. As the number of threads increases, the throughput of EXT4-oF becomes up to 13% higher than that of the EXT4(Client-Server) because EXT4-oF reduces the number of network hops and avoids the TCP/IP communication overhead.

In Figure 4(b), we present the CPU utilization results when 32 threads sequentially read a 2 GB file each. In EXT4(Client-Server), the server's CPU utilization (denoted as EXT4(Server)) is as high as 300%, even though the server only receives data from the storage node and forwards it to the client node. In contrast, EXT4-oF does not consume any CPU cycles on the server. On the client side, the CPU utilization of EXT4(Client-Server) (denoted as EXT4(Client)) is also twice as high as that of EXT4-oF.

Unlike the classic client-server model, EXT4-oF incurs file open overhead as EXT4-oF must validate the cached data blocks in the parent directory if the requested file is not found in the file system cache. In the experiments, we break down the latency of `open()` when a new file is created on a remote node. On average, it takes about 174 μsec to read an updated data block of the parent directory and about 250 μsec to read an inode table block. Thus, `open()` takes about 487 μsec to return a file descriptor. If a file is created on a local node, `open()` takes about 60∼80 μsec in EXT4, i.e., invalidating cached file metadata increases the latency of `open()` by up to 8×. However, we note that the file metadata invalidation overhead only incurs when a new file is created on a remote node, and the file is accessed for the first time. Furthermore, considering that EXT4-oF is designed for applications that read large immutable files with WORM semantics, the file open overhead does not significantly affect the overall IO throughput, as we demonstrate in Figures 4(a).

C. Performance Evaluation of MinIO-oF

Finally, we evaluate the performance of MinIO-oF using YCSB. For the experiments, we populated the database with 100,000 records containing random string data of approximately 2MB each, resulting in a total data size of 200 GB. Then, we submit 100,000 read and write queries for each workload.

Figure 5 shows the read throughput (YCSB workload C with Zipfian key distribution) of MinIO and MinIO-oF as the number of clients increases. As the number of clients increases, remote reads occur more frequently, and MinIO-oF benefits from avoiding TCP/IP communication, resulting in throughput up to 1.4× higher than MinIO.

In Figure 6, we present the throughput of each YCSB workload with 64 client threads when the key distribution is uniform. Overall, MinIO-oF outperforms MinIO in processing remote reads. and the performance difference between MinIO-oF and MinIO is greater in the uniform distribution where cache locality is low, compared to the Zipfian distribution, shown in Figure 5. However, the throughput value is lower in a uniform distribution workload compared to Zipfian, and this is due to the lower locality.

VI. Conclusion

In this work, we present EXT4-oF to explore an opportunity of sharing NVMe-oF SSDs across multiple distributed nodes without inter-node communication. EXT4-oF effectively reduces the overhead of data transfer and improves the IO throughput by up to 17% while guaranteeing cache coherence and file system consistency for immutable data through NVMe-driven lazy cache coherence protocol.

We conduct case studies with WORM applications. In particular, we develop MinIO-oF that eliminate the need for TCP/IP communication during remote reads. In our performance study, we show that EXT4-oF can improve lightweight and scalable MinIO's remote read performance by up to 1.4×.

VII. Acknowledgement

This work was supported by Samsung Electronics, and also in part by IITP grant funded by the Korea government (MSIT) (No.2021-0-00862). Tuan Anh Nguyen, Hyeongjun Jeon, and Daegyu Han made equal contributions to this work. The corresponding author is Beomseok Nam.

References

[1] NVM Express over Fabrics. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf.

[2] NVMe-oF JBOFs. https://nvmexpress.org/wp-content/uploads/NVMe-202-1-Part-1-JBOFs_Final.pdf.

[3] Storage System using NVMe over Fabric SSD-Based Ethernet JBOF. https://www.flashmemorysummit.com/Proceedings2019/08-08-Thursday/20190808_NVMF-302B-1_Chung.pdf.

[4] The Future of Software-Defined Storage. https://www.snia.org/sites/default/files/SDC15_presentations/gen_sessions/RichardMcDougall_Software_Defined_Storage_Look-Like.pdf.

[5] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 4th USENIX conference on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[7] Mark Fasheh. OCFS2: The Oracle Clustered File System, Version 2. In *Proceedings of the 2006 Linux Symposium*, volume 1, pages 289–302. Citeseer, 2006.

[8] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR)*, pages 1–9, 2017.

[9] Daegyu Han and Beomseok Nam. Improving Access to HDFS using NVMeoF. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–2. IEEE, 2019.

[10] Pat Helland. Immutability Changes Everything. *Communications of the ACM*, 59(1):64–70, 2015.

[11] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 756–771, 2021.

[12] Wonbae Kim, Young-ri Choi, and Beomseok Nam. Mitigating YARN Container Overhead with Input Splits. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 204–207, 2017.

[13] Nancy P Kronenberg, Henry M Levy, and William D Strecker. VAXcluster: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems (TOCS)*, 4(2):130–146, 1986.

[14] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[15] Dave Minturn and J Metz. Under the Hood with NVMe over Fabrics. In *Ethernet Storage Forum. SNIA*, 2015.

[16] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[17] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop Yarn: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[18] Tom White. *Hadoop: The Definitive Guide*. " O'Reilly Media, Inc.", 2012.

[19] Steven Whitehouse. The GFS2 Filesystem. In *Proceedings of the Linux Symposium*, pages 253–259. Citeseer, 2007.

[20] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. Efficient User-Level Storage Disaggregation for Deep Learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.