

Improving Remote File Access in Distributed Object Stores by Decoupling Metadata and Data Paths using NVMe-oF

Daegy Han, *Dept. of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, South Korea*

Sungho Moon, *Dept. of Computer Science and Engineering, Sungkyunkwan University, Suwon, South Korea*

Kyeungpyo Kim, *GlueSys Co., Ltd., Anyang, South Korea*

Sung-Soon Park, *GlueSys Co., Ltd., Anyang, South Korea*

Beomseok Nam, *Dept. of Computer Science and Engineering, Sungkyunkwan University, Suwon, South Korea*

Abstract—

Storage network protocols such as NVMe-oF operate below the file system layer. Therefore, even when NVMe-oF allows storage volumes to be shared across the network, compute nodes cannot access remote files managed by another node's file system without a cluster file system. In conventional distributed systems, accessing files owned by a remote node requires communication with the remote node via RPC. The remote node then retrieves the data from a disaggregated storage node and transfers it to the requesting node. To reduce redundant network traffic, this study proposes RDIO, which separates RPC-based remote data access into two distinct planes: a metadata plane for file mapping and a data plane for direct access to remote storage. The data plane ensures data flows only through the storage network. We integrate RDIO into MinIO and show that RDIO significantly improves performance by reducing remote data movement between nodes.

With the increasing demand for scalable storage services and efficient resource utilization, modern data centers are rapidly transitioning from monolithic server architectures where compute and storage resources are tightly coupled to disaggregated storage architectures [1]. State-of-the-art storage network protocols such as NVMe-oF (NVMe over Fabrics) enable physical separation of compute and storage resources, and allow multiple hosts to dynamically share storage resources [2].

However, the storage network protocol is a block-level protocol that operates below the file system layer. Therefore, compute nodes cannot share access to files owned by remote compute nodes without file system support. This limitation prevents distributed object stores from leveraging the distance connectivity of NVMe-oF. That is, distributed object stores need to use RPC communication to access remote files. This RPC communication between compute nodes results in unnecessary data transfer over the network.

Specifically, data transmission between the remote compute node and the storage node is followed by the transmission to the local compute node. This sequence of data transfers doubles the I/O latency, and we refer to this problem as the *double transfer* problem.

To resolve the double transfer problem, we propose RDIO, a novel remote file access mechanism to enable direct access to storage volumes managed by remote nodes without using conventional shared-disk file systems. RDIO consists of *metadata plane* and *data plane*. The metadata plane retrieves file mapping information from remote nodes, and converts file offsets into physical block addresses. Using the physical block addresses, the data plane directly reads and writes the disaggregated storage volume via NVMe-oF. This ensures file blocks are transmitted over the network only once such that both network and I/O stack overhead are reduced.

Key contributions of this study are as follows:

- First, this study introduces a novel approach for accessing files on remote nodes by leveraging the distance connectivity of NVMe-oF. RDIO separates metadata and data planes and minimize

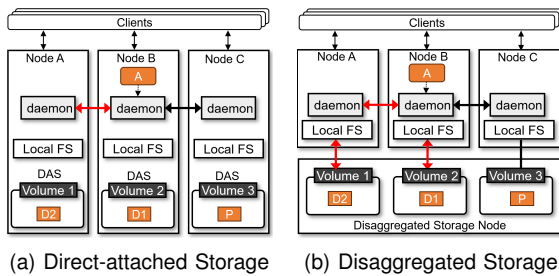


FIGURE 1. Distributed Object Store on Different Storage Architectures

RPC communication between compute nodes.

- Second, this study presents *parity-free decoding* and *deferred parity encoding* optimizations, which improve erasure coding workflows by enabling asynchronous parity block encoding while ensuring data availability even when remote nodes are inaccessible.
- Third, we implement RDIO in MinIO, a popular distributed object store, demonstrating its potential to improve performance in distributed object storage systems.

Distributed Object Store and Disaggregated Storage

Distributed Object Store

Distributed object stores, such as MinIO, are widely used in modern data centers to efficiently handle large volumes of unstructured data. To ensure high availability and fault tolerance, these systems typically employ replication or erasure coding [3].

Designed for a shared-nothing architecture, distributed object stores utilize single-node file systems, such as EXT4 and XFS, instead of shared-disk file systems. Figure 1(a) shows an example of a distributed object store using erasure coding on three nodes with DAS, where an object (A) is encoded into two partition blocks (D1 and D2) and one parity block (P). When reading the object, if there is no failure, only the two partition blocks (D1 and D2) are accessed and merged to restore the object. If partition blocks are corrupted or one of the nodes that contain partition blocks does not respond, the parity block is used to decode and restore the object.

Storage Disaggregation

Figure 1(a) shows the architecture of traditional monolithic server with DAS (Direct-attached Storage). Since compute and storage resources are tightly coupled, it is difficult to scale each resource individually, which leads to over-provisioning of resources and low resource utilization. To overcome this issue, modern data centers have shifted towards disaggregated storage architectures, where storage resources are physically separated from compute nodes [1], as shown in Figure 1(b). Storage network protocols, including NVMe-oF and iSCSI, allow disaggregated storage devices to be accessed as if they were local block devices [2]. NVMe-oF is the state-of-the-art storage protocol that transmits block-level I/O requests (NVMe commands) over the network. With RDMA support, NVMe-oF reduces data copy overhead and minimizes CPU usage [2].

Shared Access to Disaggregated Storage Storage disaggregation allows multiple compute nodes to access storage resources simultaneously. However, a storage network protocol alone does not resolve conflicts that occur when multiple compute nodes compete for the same block [4], [5]. For example, allocating a block that is already in use by another compute node causes conflicts. To resolve such conflicts and enable shared access to files on remote storage, shared-disk file systems have been used.

General-purpose POSIX shared disk file systems run daemon processes on each compute node to resolve conflicts and prevent inconsistent file accesses. However, due to these consistency checks, shared-disk file systems suffer from non-negligible overhead and exhibit lower performance compared to single-node file systems.

Double Data Transfer Problem

When deployed on disaggregated storage, accessing a file owned by a remote node in distributed object storage causes the file data to traverse the network twice for a single I/O operation. Specifically, suppose a data node requests file data from a remote data node, which commonly occurs in distributed object stores that use erasure coding. The remote node retrieves the file data from the disaggregated storage node, using the storage network. Then, the remote node sends the file data to the requesting node, using the compute network. This double data transmission consumes significant CPU cycles on both data nodes and increases network traffic [4], [5].

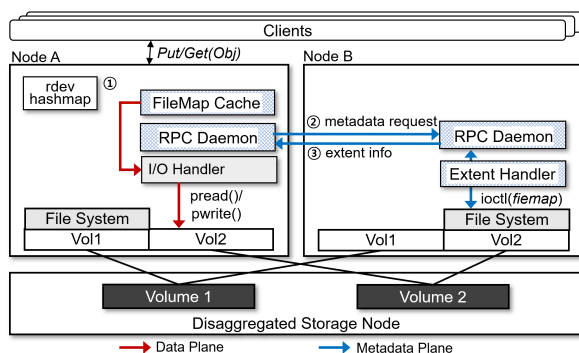


FIGURE 2. RDIO Architecture and I/O Path

Design and Implementation of RDIO

Figure 2 shows the architecture of RDIO (Remote Direct I/O). Each node has its local disk volume and accesses it using a local file system. These disk volumes are shared by remote nodes through a storage network protocol. We refer to these volumes as remote volumes. To avoid any potential file system inconsistency, file system-level access to remote volumes is not allowed. In the next sub-section, we detail how RDIO supports remote file access.

Remote Direct I/O Path

Rdev Hashmap. RDIO manages metadata about remote volumes in the `rdev hashmap`. Specifically, this hashmap maps remote nodes' file system mount points to their corresponding remote volumes on the local node.

Write. In the legacy system, in order to write a file to a storage volume managed by a remote node, the write request and data were sent to the remote node and written. In contrast, RDIO uses the metadata plane that sends an RPC request to the remote node (Node B in Figure 2) for file allocation, including metadata such as the file path and data size. The RPC stub on the remote node triggers the Extent Handler, which uses the `fallocate()` system call to allocate the file. The file mapping information for the allocated file is retrieved using the `ioctl(fiemap)` and returned to the requester node (Node A). Once the metadata plane for remote write is handled, the requester node performs the data plane by converting the file offset to an LBA (Logical Block Address) and executing the `pwrite()` system call on the raw block device.

Read. For read operations, the requester node sends an RPC request to retrieve the file mapping information from the remote node, which queries the

file mapping information from the file system using the Extent Handler and returns it to the requester node. Then, similar to the write process, the requester node uses the file mapping information and issues the `pread()` system call to directly read data from the disaggregated raw block device. It is noteworthy that unless there is significant memory pressure, the `ioctl(fiemap)` does not result in disk I/O since the file mapping is like to have been cached in the remote node's file system cache.

RDIO provides shared access to files on remote storage while still leveraging the existing Linux native file system. The remote data transfer process in RDIO is divided into metadata and data planes, ensuring that data is transmitted only once across the network fabric, unlike the traditional RPC-based access. Specifically, while data is transmitted through the storage network protocol, its metadata (e.g., file mapping information) is retrieved between nodes via RPC communication. Despite RPC communication on the metadata plane, since metadata is much smaller than the actual data, RDIO can reduce CPU usage and network traffic during remote data transfer.

Improving Metadata Plane for File Mapping

Every read or write request requires file mapping information, and each I/O operation must communicate with the remote node via RPC to access the file system's metadata. Although the size of the metadata is not large, there may be issues with increased I/O latency. To address this problem, RDIO implements *FileMap Cache*, a cache for file mapping information. The FileMap Cache eliminates one network hop in the metadata plane, allowing only the data plane to be performed through cached entries. As a result, when a cache hit occurs in Node A's FileMap Cache, the RPC request for file mapping is skipped, and the data can be directly read from the remote volume.

Entries in the FileMap Cache consist of the file path and file mapping information. As a metadata cache that does not store actual data, it has a small memory footprint, making it suitable for environments handling large datasets. Additionally, the FileMap Cache takes advantage of temporal locality for frequently re-accessed entries by applying the LRU cache replacement policy.

Cache consistency in distributed systems is crucial to ensure that all nodes have a consistent view of the data. On the other hand, there exist various modern applications that allow for weak cache consistency models. In distributed object stores such as MinIO, which consider the immutability of objects, complex cache consistency mechanisms used in generic shared disk file systems are not necessary. If objects

are mutable, their file mapping information stored in the FileMap Cache may become stale. This poses a risk of reading incorrect data from the remote storage. Therefore, the cached entries need to be invalidated when the object's file mapping is changed by other nodes as in conventional shared disk file systems.

MinRDIO: Integrating RDIO into Distributed Object Storage

We integrate RDIO into MinIO (RELEASE.2022-05-04T07-45-27Z) to verify the applicability of RDIO and name it MinRDIO. In this section, we first describe how RDIO is specifically applied in MinRDIO, followed by a discussion of the novel resilience mechanism enabled by RDIO.

Putting it All Together

Write. MinIO uses erasure coding to split a new object into encoded blocks (i.e., partition and parity blocks) and then distributes them across multiple nodes along with the metadata file (i.e., `xl.meta`) using consistent hashing. When writing encoded blocks to remote nodes, the remote write path of RDIO is used. For metadata files that are considerably smaller (e.g., a few hundred bytes) than the encoded blocks, the existing RPC-based remote write path is used as is. Additionally, the RPC requests for file mapping that occur in RDIO's remote write path are performed asynchronously during object encoding. This achieves an effect similar to executing only the data plane while hiding the processing in the metadata plane.

MinIO ensures data persistence by flushing encoded blocks and their associated metadata files to disk. MinRDIO aims to achieve the same goal, but since remote volumes are not mounted on the local node, file system operations such as `fdatasync()` cannot be directly invoked for encoded blocks. Note that MinIO's write operation consists of two phases: writing the encoded blocks first, followed by the metadata files. Similarly, MinRDIO uses RDIO to store the encoded block and then writes the metadata file to a remote node. To ensure the persistence and crash consistency, the file path to the encoded block is provided to the remote node such that the encoded block is flushed to disk first using `fdatasync()`, followed by the metadata file.

Read. Since MinIO uses consistent hashing to determine the location of partition blocks, it first reads all metadata files to identify their locations and then accesses the corresponding partition blocks. When reading metadata files and partition blocks from remote

nodes, both use the remote read path of RDIO, and the obtained file mappings are cached in the FileMap Cache. If the file mapping is cached in the FileMap Cache, the partition block or metadata file is read from the remote volume without metadata plane RPC requests.

Direct I/O vs. Buffered I/O. Linux file systems typically use buffered I/O with the page cache to minimize disk I/O. They also support direct I/O, which bypasses the page cache, allowing applications to choose between these I/O options.

When using direct I/O, both MinIO and MinRDIO bypass the page cache of both local and remote compute nodes. As a result, both incur the same storage access cost for each I/O request. However, MinIO consumes additional network bandwidth between compute nodes, whereas MinRDIO avoids this by enabling direct access to the remote shared disk volume.

Although MinRDIO is unable to exploit buffered I/O for write operations due to the constraints of using raw devices, the cost of remote I/O remains comparable to MinIO with buffered I/O, as both need to flush data to storage for persistence. Since Linux treats block devices as files, raw devices can benefit from buffered I/O, allowing MinRDIO to leverage the local node's page cache for read operations. It reduces unnecessary network round trips compared to MinIO, which relies on the remote node's page cache.

Resilience Mechanism in MinRDIO

In large-scale distributed systems, various failures, e.g., software, hardware, and network partitioning, occur frequently. To mitigate these issues, distributed object stores implement resilience mechanisms such as erasure coding and replication. However, these come with additional computational and space overhead.

Storage nodes contain multiple storage devices, and if a single storage node holds both data and parity blocks, it can become a single point of failure. This study does not address storage node failures, nor does it consider cases where partition blocks are corrupted. These failures need to be addressed by replication or other fault-tolerance mechanisms, which are orthogonal to the parity-based optimization that we describe below.

Parity-free Decoding. In the event of a compute node failure, vanilla MinIO restores partition blocks owned by the failed node by reading parity blocks.

In contrast, MinRDIO implements *Parity-free Decoding*, which reconstructs the object without reading parity blocks. Without reading the parity block, MinRDIO can access lost partitions from other available

compute nodes, allowing the original object to be recovered without the need to read or decode parity blocks.

However, although parity blocks are not needed, file mapping information from the failed node is still needed. If the file mapping is available in the FileMap Cache in the local node, MinRDIO reads the file mapping from the FileMap Cache. If the file mapping is not found in the local cache, MinRDIO mounts the file system of the remote storage volume on the local node. Then, it retrieves the file mapping and read the partition block. Alternatively, MinRDIO can fall back to the legacy method of decoding parity blocks to recover the object.

Deferred Parity Encoding. MinRDIO can directly access partition blocks managed by remote nodes even when the nodes are unavailable. That is, the availability of remote nodes is not relevant to the availability of partition blocks, and thus MinRDIO minimizes the need for parity blocks.

In disaggregated storage systems, multiple block devices are often configured using RAID to create logical disk volumes, such that even when a block device fails, its partition blocks can be restored. In contrast, even if RAID is used in disaggregated storage systems, the vanilla MinIO requires parity blocks because partition blocks can be accessed only through their corresponding nodes.

Since parity blocks at the MinRDIO layer are not essential for data availability, we propose *Deferred Parity Encoding*. In this approach, MinRDIO writes partition blocks synchronously while writing parity blocks asynchronously.

Although parity blocks are not essential in MinRDIO, we let MinRDIO use parity blocks for legacy support. Additionally, having parity blocks at the MinRDIO layer helps recover from various scenarios such as where partition blocks are distributed across multiple disaggregated storage nodes, and some of those nodes fail simultaneously.

Experiments

Experiment Setup

We conduct a performance evaluation using a six-node cluster (one client node, four compute nodes, and one storage node). Each node is equipped with two 10-core Xeon Gold 5115 processors (2.4 GHz/14 MB) running on Linux kernel 5.3.0-24-generic with hyperthreading enabled and 64 GB DRAM. The storage node has two 8-core Xeon Silver 4215 processors (2.50 GHz/11 MB), 64GB DRAM, and eight Samsung PM983 U.2

NVMe SSDs. All nodes are connected via a 56 Gbps Mellanox ConnectX-4 Infiniband interconnect.

Following the (k,r) erasure coding construction, we use (6, 2) configuration, consisting of six data partitions and two parity partitions. Each of the four nodes is configured with two NVMe volumes, each mounted with an XFS.

We compare the performance of MinRDIO to MinIO, both configured to use direct I/O. For MinRDIO, we evaluate the performance with and without FileMap Cache for read queries. In the former case, the FileMap Cache capacity is set to accommodate 4 M objects, enough to cache all file mappings for the dataset and cache warmup is performed before query execution. The latter is named MinRDIO-FMC.

Evaluation of RDIO

First of all, we quantitatively measure and analyze the effects of RDIO using REST-API-based microbenchmarks. To measure throughput, we use 40 threads to execute requests on datasets of 40 GB, varying the object size, while latency analysis is measured single-threaded on 32 KB objects. CPU consumption and network traffic are shown for 1024 KB objects.

Throughput and Latency. Figures 3(a) and 3(b) show that MinRDIO achieves up to 11% higher write throughput and up to 48% higher read throughput than MinIO by avoiding the double transfer. Compared to MinRDIO-FMC, MinRDIO shows 9–23% higher throughput because RPC communication for file mapping retrieval is eliminated. As the object size increases, the I/O time increases. Accordingly, the relative overhead of RPC in obtaining file mapping is reduced.

When the object size exceeds 16 MB, MinRDIO exhibits slightly lower throughput than MinIO. Unlike MinIO, which allows remote compute nodes to process I/O for remote blocks, MinRDIO handles all remote I/Os on the local compute node. For small objects, the benefit of reduced network traffic outweighs the impact of I/O contention. However, for large objects, increased I/O contention on the local compute node negates the advantage of RDIO. This limitation could be mitigated by dynamically enabling or disabling RDIO based on the compute node's I/O load. We leave this as future work.

Despite the benefits of RDIO, the improvement in write throughput is not as significant as that in read throughput. To understand the reason for the performance difference, we analyze the latency as shown in Figures 3(c) and 3(d).

Surprisingly, the actual latency of writing encoded blocks (`writeBlocks`) in MinRDIO is 85% lower than

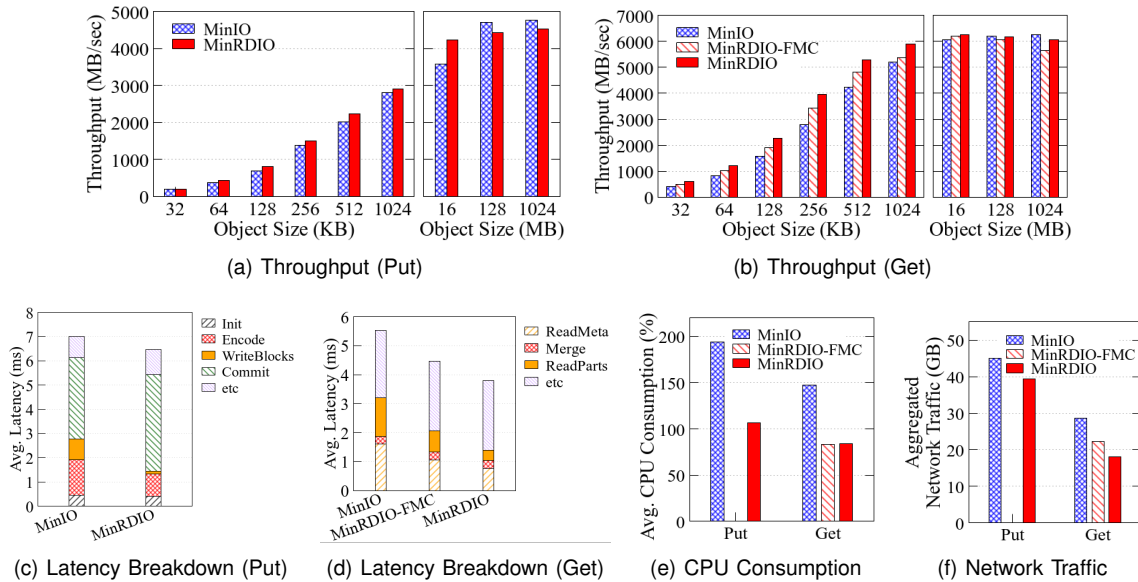


FIGURE 3. Quantification Results

that of MinIO. However, the performance improvement is hidden because a significant portion of the total work time is spent on writing metadata files, committing (`Commit`) which changes the file path of encoded blocks to the actual bucket name, and encoding objects (`Encode`). MinRDIO shows slightly higher latency than MinIO because it performs `fdatasync()` on encoded blocks during commit. On the other hand, the reason why the encoding latency of MinRDIO is reduced is because other tasks are performed asynchronously, unlike MinIO which synchronously waits for completion during the object encoding.

MinRDIO shows 62% lower read latency than MinIO by affecting metadata (`ReadMeta`) and partition blocks (`ReadParts`). Although the overhead of the object storage interface related to returning objects to the client accounts for a significant portion of the total operation time, the performance improvement effect of RDIO is still remarkable.

CPU Consumption and Network Traffic. Figure 3(e) shows that MinRDIO consumes 12% less CPU when writing objects and 37% less CPU when reading objects compared to MinIO. This is because MinRDIO only transfers data to remote volume via a storage network. Write operations consume more CPU resources than read operations due to the need for object encoding and the writing of both data and parity blocks.

As shown in Figure 3(f), MinIO generates 1.82x and 1.76x more traffic than MinRDIO for write and

read operations, respectively. This is because data is transferred redundantly between compute and storage nodes as well as between compute nodes. Considering the network traffic difference between MinRDIO-FMC and MinRDIO, the network traffic used by the metadata plane RPC for file mappings is not that much.

Evaluation of Scalability

The experiments shown in Figure 4 show latency-throughput curves comparing the performance of MinRDIO with MinIO for YCSB workloads, varying the number of client threads from 16 to 256. For Load, we measure the average latency for write operations, and for the other workloads, we measure the average latency for read operations. For workload C (100% reads), we compare the performance with the optimal case (`Optimal`), which performs local read operations by routing remote read requests to mounted remote volumes on the local node. Referring to recent studies [6], [7], we set the object size to 32 KB–1 MB, and the key-value size and request distribution follows a Zipfian distribution. The dataset is filled with 800,000 records (about 80 GB) each containing random string data. Then, 800,000 read and write queries are submitted for each workload.

As the number of clients increases, MinRDIO has lower latency and higher throughput overall than MinIO. This is because remote access becomes more frequent and MinRDIO avoids TCP/IP communication in the data path. Due to the expensive write process

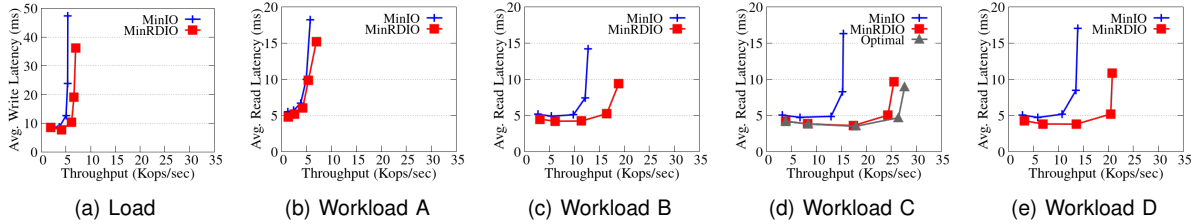


FIGURE 4. Scalability Results: Latency-Throughput Analysis

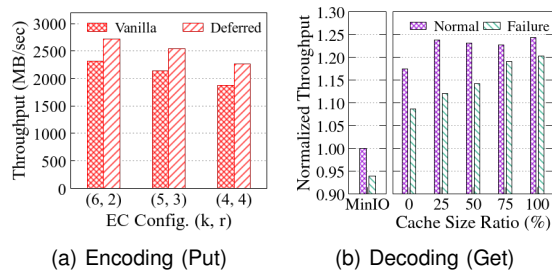


FIGURE 5. Resilience Mechanism Performance

of MinIO, throughput gradually saturates and latency increases from 64 threads for write-intensive workloads compared to read-intensive workloads. Nevertheless, for write-intensive workloads Load and Workload A (50% write and 50% read), MinRDIO achieves $1.29\times$ and $1.21\times$ higher throughput, respectively, by reducing CPU consumption and network traffic. In read-intensive workloads, significant performance improvements are observed, especially for read-only workload C, MinRDIO shows $1.66\times$ higher throughput and 33% lower latency than MinIO at 256 threads. MinRDIO achieves performance comparable to Optimal overall. However, at 256 threads, internal operations such as the FileMap Cache impact parallelism, resulting in approximately 7% lower throughput.

Evaluation of Resilience Mechanism

Finally, we evaluate the performance of MinRDIO's resilience mechanism. Using microbenchmarks, we query 10,000 4MB objects (about 40GB) with 40 threads.

Figure 5(a) shows the impact of Deferred Parity Encoding varying on the erasure coding configuration (k, r), assuming RAID-5 storage. We compare Deferred Parity Encoding (denoted as Deferred) with the default encoding scheme of MinRDIO (denoted as Vanilla). Deferred can take advantage of the ability to read all partition blocks stored in a storage node

even if a failure occurs, thereby reducing the object encoding time by asynchronously storing parity blocks. As a result, the throughput of Deferred is up to 20% higher than that of Vanilla. As r increases from 2 to 4, the write throughput slightly decreases, which is due to the larger size of the partitions.

In the experiment in Figure 5(b), we evaluate the impact of Parity-free Decoding by varying the ratio of cache size to the number of objects in the FileMap Cache. After populating the dataset, we induce a failure in one of the nodes except the one that received the client's request. The Normal scenario represents no failure, while the Failure represents a scenario where the object storage process is terminated due to a system failure and the node is inaccessible. Due to the consistent hashing that randomly distributes partitions, some objects may not be affected by the object decoding because their parity blocks are stored on the failed node. The results of this experiment are normalized to the throughput in the Normal scenario of MinIO.

When a failure occurs, MinIO's Failure throughput decreases by 7% compared to the Normal. This decrease is because the object reconstruction involves decoding parity blocks to recover missing partition blocks. On the other hand, MinRDIO can return the object to the client using only the partition blocks without performing decoding if the file mapping of the partition block is cached, achieving up to $1.27\times$ higher throughput than MinIO. When a cache miss occurs in the FileMap Cache for a partition block, we set an option to fall back to the default decoding mechanism. Therefore, as the cache size ratio decreases, the throughput decreases due to expensive decoding and causes a performance gap with Normal. However, MinRDIO achieves higher throughput than MinIO because it utilizes remote read paths even when reading parity blocks.

Conclusion

In this study, we explored practical and novel designs for shared storage systems in disaggregated storage. Due to the lack of file-level shared access support in storage network protocols, distributed object stores rely on RPC-based access, even when storage volumes in disaggregated storage are shareable. To address this challenge, we propose RDIO, which enables shared access to remote data by leveraging file mapping information while making use of existing Linux native file systems. To showcase its applicability, we implement MinRDIO and design a new resilience mechanism that takes advantage of RDIO. Our performance evaluation shows that MinRDIO improves I/O throughput while reducing CPU consumption and network traffic by eliminating redundant remote data transfer.

Acknowledgement

This work was supported by IITP grant funded by the Korea government (MSIT) RS-2021-II210862 (2021-0-00862) and RS-2024-00461572.

REFERENCES

1. P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 249–264.
2. "NVM Express over Fabrics," [n.d.], <https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf>.
3. "Minio: The Object Store for AI Data Infrastructure," [n.d.], <https://github.com/minio/minio>.
4. T. A. Nguyen, H. Jeon, D. Han, D.-H. Bae, Y. J. Yu, K. Kim, S. Park, J. Jeong, and B. Nam, "NVMe-Driven Lazy Cache Coherence for Immutable Data with NVMe over Fabrics," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 2023, pp. 394–400.
5. H. Li, S. Jiang, C. Chen, A. Raina, X. Zhu, C. Luo, and A. Cidon, "RubbleDB: CPU-Efficient Replication with NVMe-oF," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 689–703.
6. F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/o characterization and performance evaluation of beegfs for deep learning," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.
7. O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's Time to Revisit LRU vs. FIFO," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.