ArkFS: A Distributed File System on Object Storage for Archiving Data in HPC Environment

Kyu-Jin Cho
Seoul National University
Seoul, South Korea
bori19960@snu.ac.kr

Injae Kang
Seoul National University
Seoul, South Korea
abcinje@snu.ac.kr

Jin-Soo Kim
Seoul National University
Seoul, South Korea
jinsoo.kim@snu.ac.kr

***Abstract*—As the burst buffer is being widely deployed in the HPC (High-Performance Computing) systems, the distributed file system layer is taking the role of campaign storage where scalability and cost-effectiveness are of paramount importance. However, the centralized metadata management in the distributed file system layer poses a scalability challenge. The object storage system has emerged as an alternative thanks to its simplified interface and scale-out architecture. Despite this, the HPC communities are used to working with the POSIX interface to organize their files into a global directory hierarchy and control access through access control lists.**

In this paper, we present ArkFS, a near-POSIX compliant and scalable distributed file system implemented on top of the object storage system. ArkFS achieves high scalability without any centralized metadata servers. Instead, ArkFS lets each client manage a portion of the file system metadata on a per-directory basis. ArkFS supports any distributed object storage system such as Ceph RADOS or S3-compatible system with an appropriate API translation module. Our experimental results indicate that ArkFS shows significant performance improvement under metadata-intensive workloads while showing near-linear scalability. We also demonstrate that ArkFS is suitable for handling the bursty I/O traffic coming from the burst buffer layer to archive cold data.

***Index Terms*—High-performance computing, Distributed file system, Object storage**

I. Introduction

Traditionally, various distributed file systems have been developed to offer high bandwidth/IOPS and large capacity for the High-Performance Computing (HPC) environment. However, the burst buffer equipped with fast storage devices such as Solid-State Drives (SSDs), is quickly replacing distributed file systems due to its higher performance. As the burst buffer is being widely deployed in the HPC systems, the distributed file system layer is shifting its role as capacity tier or *campaign storage* [19] where scalability and cost-effectiveness are of paramount importance.

In general, the traditional distributed file systems are based on the architecture in which several storage nodes are grouped together to form a storage cluster and a single centralized metadata server manages file system metadata. This centralized metadata server architecture can guarantee file system consistency easily, but it severely hampers scalability. Therefore, recent distributed file systems aggregate metadata servers to construct a metadata server cluster, and distribute global namespace hierarchy across a set of metadata servers either statically or dynamically to achieve highly scalable metadata management performance [21], [35], [38]. However, these distributed file systems with dedicated metadata servers are still not scalable, complex, and also require constant maintenance. This scalability issue is becoming more and more challenging as the metadata throughput is critical for large-scale HPC environment.

For storing exascale data and its associated metadata in a scalable and efficient manner, distributed object storage systems such as Amazon S3 [3] and Ceph Object Storage [17], [38] have emerged as an alternative to distributed file systems. Distributed object storage systems store data in a key-value form and guarantee high durability and reliability by means of replication and erasure coding mechanisms. In addition, unlike the hierarchical namespace of a file system, the flat namespace used in the object storage allows users to easily scale out the overall performance and capacity. Such high reliability, cost-effectiveness, and scalability open up a new opportunity where scalable object storage systems can be utilized for the HPC storage system.

Object storage systems use REST (Representational State Transfer)-based operations such as `GET`, `PUT`, and `DELETE`. These simple operations work great for applications that are newly written to use this interface for a specific object storage system. However, legacy applications cannot be run directly on the object storage because they are written using the POSIX interface. More specifically, many HPC applications rely on the global directory hierarchy where the accesses to files and directories are controlled via access control lists [26]. For this reason, the HPC community still prefers distributed file systems that support the POSIX interface or its extensions [34], [39].

In this paper, we introduce ArkFS, a near-POSIX, scalable distributed file system implemented on top of the object storage system with client-side metadata service. ArkFS is designed for an environment where data needs to be quickly transferred to and from the burst buffer layer for archiving purposes by a small number of administrator processes in the background, rather than an environment that is regularly accessed by general users. Considering this controlled environment, we propose an architecture that can quickly serve metadata and data operations between the burst buffer and the

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00038

underlying object storage system through the POSIX interface. To better support the aforementioned archiving role under the controlled environment, ArkFS has the following three design goals:

- **Support for various object storage backends.** ArkFS provides a file system interface on top of any distributed object storage system by simply registering their REST APIs. ArkFS represents both file system metadata and data as objects and performs translation between POSIX APIs and REST APIs.
- **High scalability with client-driven metadata service.** ArkFS supports parallel metadata operations on a per-directory basis with a client-driven metadata service. In ArkFS, clients actively participate in file system metadata management for individual directories without the centralized metadata server. This enables ArkFS to achieve highly scalable metadata performance.
- **Near-POSIX compatibility.** ArkFS covers most of the standards specified by POSIX. These include POSIX file system APIs, a global directory hierarchy with access control lists and POSIX consistency semantics on a shared environment.

To demonstrate the usefulness of ArkFS under the heavy I/O traffic coming from the burst buffer, we perform several experiments and compare their results with those obtained from other file systems. Our evaluations show that ArkFS is superior to MarFS and it can achieve up to 24.86x higher throughput than CephFS under metadata-intensive workloads. Also, ArkFS achieves better scalability in metadata management than other file systems. Moreover, ArkFS outperforms S3-based file systems in terms of the `READ`/`WRITE` bandwidth during large file I/Os and demonstrates competitive performance compared to CephFS. Using the realistic scenario with the *tar* tool, we also show that ArkFS can effectively support the archiving role under the controlled environment.

The rest of the paper is organized as follows. We present our motivation in Section II. Section III discusses the design and the fault tolerance of ArkFS, respectively. Section IV shows the experimental results and Section V concludes the paper.

II. Related Work and Motivation

A. Distributed File System Architectures

Distributed file system (DFS) provides a hierarchical namespace of files and directories upon physical machines that are loosely-coupled by a network. Most of the commonly used DFSs are composed of a set of storage nodes for storing data and a dedicated server for metadata management [22], [24], [35]. However, the DFS architecture with a centralized metadata server leads to a scalability problem because all of the clients' requests are directed to this single component. Also, a single metadata server degrades the availability of the file system because it becomes a SPOF (Single Point of Failure). Thus, state-of-the-art DFSs usually deploy multiple metadata servers and construct a metadata server cluster with its own metadata load balancing policy [29], [35], [38].

To address metadata-intensive workloads, recent research have suggested middleware solutions that exploit client resources to manage metadata, in a layer between clients and distributed file systems [28], [30], [40], [41]. They allow each application to manage a subset of file system metadata in a private namespace during its execution. As there is no need to access metadata servers frequently to perform metadata operations, this approach exhibits higher throughput in metadata-intensive or checkpointing workloads.

B. File Systems over Object Storage

Given that the burst buffer layer is responsible for providing fast I/O in HPC systems, scalability and cost-effectiveness become main concerns of the distributed file system layer. For this reason, object storage, known for its high scalability and reliability, emerges as a viable alternative to DFS as campaign storage. Distributed object storage, such as Ceph RADOS [38], Amazon S3 [3], OpenStack Swift [17], and Intel DAOS [25], are convenient to use with their pre-defined REST APIs. However, the HPC community often prefers to use the POSIX file system interface because it provides many useful features such as directory hierarchy, access control, symbolic links, etc.

Ceph provides the file system interface through CephFS that works on top of a reliable object storage layer called RADOS [38]. There are also Amazon S3-based file systems such as S3FS [13] and goofys [6]. These file systems work on top of object storage and convert POSIX APIs from legacy applications into object storage's REST APIs. Therefore, existing applications can utilize the benefits of object storage without any application-level change.

C. Motivation

Despite the numerous file systems proposed to leverage the advantages of object storage, they are not well-suited for the controlled environment where large volumes of data are moved to and from the burst buffer layer by daemons or administrators. We have identified the two most challenging issues when the HPC community wants to use the object storage for their cold storage tier or campaign storage.

Challenge 1) Supporting the POSIX-compliant Interface

POSIX defines the Application Programming Interfaces (APIs) such as `open()`, `read()`, `write()`, etc., and their consistency semantics for the compatibility across different file systems [9]. POSIX consistency semantics specifies what is and is not guaranteed to happen when a specific API call is made. It contains a level of consistency for shared variables and atomicity of file system operations.

The HPC community heavily depends on the support of POSIX API. HPC users are accustomed to the hierarchy of directories and files where they can organize their data according to projects and dates, and control the accesses using per-directory or per-file access control lists (ACLs). Also,

many legacy scripts and archiving/backup software work on the POSIX APIs.

However, supporting the full POSIX APIs over the object storage is never easy, because the object storage merely supports much simplified REST APIs such as `GET`, `PUT`, `DELETE`, etc. This is why well-known cloud object storage file systems such as S3FS [13], goofys [6], SVFS [15] and SwiftFS [14] provide only a limited form of POSIX APIs on top of the public object storage system. For example, S3FS is just a FUSE [33]-based wrapper layer over the Amazon S3 cloud storage that allows to access objects using the POSIX APIs. In S3FS, each S3 bucket can be mounted as a local file system. As each object is mapped to a file in S3FS, random writes or appends to files result in rewriting of the entire object. Because the object's key is treated as a full pathname, renaming of a directory leads to a situation where all the files under the directory are rewritten. In addition, permission check is not done rigorously and no coordination is performed between multiple clients mounting the same bucket. As another example, Intel DAOS [25] also provides a file system interface upon their object storage pool similar to S3FS. DAOS file system allows the DAOS storage engine to be accessed as a hierarchical POSIX namespace, but it does not support POSIX ACLs and related system calls that can change the ownership of a file.

On the other hand, recent studies proposed the *middleware* file systems which provide a file system interface between applications and a traditional DFS while preserving the latest metadata in the applications' namespaces. For instance, IndexFS [30] and BatchFS [40] let applications work in their private namespaces, and then publish modifications later as a batch job. However, they require their own dedicated metadata servers to merge batched updates and they can ensure only a relaxed consistency because one application cannot recognize the latest modifications in the other application's namespace until they are committed.

As an extension of the above file systems, DeltaFS [41] is proposed to eliminate the dedicated metadata server and allow applications to operate on a private namespace without any synchronization. However, DeltaFS does not provide a global namespace to the applications, so it can be used under limited workloads only. In contrast, Pacon [28] reemphasizes the importance of the global namespace for data sharing and file system manageability. However, it still does not fully guarantee the POSIX consistency semantics. Another relevant work is the object-centric metadata management system known as SoMeta [32], which operates between applications and object storage and presents similar design challenges as our work. Nevertheless, SoMeta does not comply with the POSIX interface as it organizes metadata using a flat namespace structure.

To summarize, all the middleware file systems fail to provide the global namespace with the full support for POSIX consistency semantics. Also, since these file systems are implemented in the form of a user-level library, the users have to rewrite their applications using the specific APIs provided by each file system. One of the design goals of ArkFS is to support a near-POSIX[1] compatibility (including both APIs and the consistency semantic) on top of the object storage system. To achieve that, we came to the conclusion that we need to keep not only the file data but also the file metadata, including inodes and directory entries, in the form of objects. These metadata should be represented as key-value data as well and stored in the object storage similar to the ordinary file data. This is where the next challenge comes in. If we have to manage all the metadata atomically, we may also need centralized metadata server(s) as in the traditional DFSs.

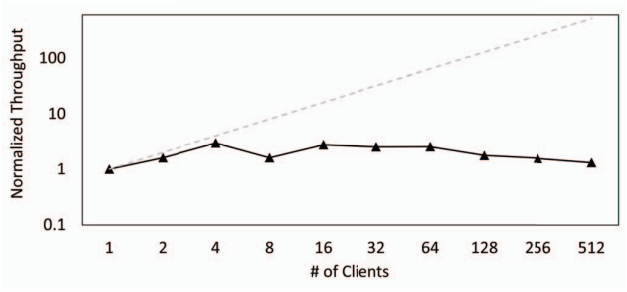


Fig. 1: **Scalability problem of a dedicated metadata server**. Massive file creations are performed while varying the number of clients up to 512. The dotted line indicates the ideal, linearly scalable performance.

Challenge 2) Scalability of Metadata Management

In the traditional DFSs, when a client tries to access the metadata of a specific file, it issues a metadata operation to the metadata server (MDS) and the MDS performs the operation atomically on behalf of the client. As we mentioned before, this centralized MDS architecture has a critical drawback in terms of scalability and availability. To achieve scalable metadata performance, several DFSs have proposed the partitioning of the file system hierarchy among a group of MDSs [36], [38]. Despite such attempts, the users are still having trouble with reaching the ideal performance due to round-trip overheads, lock contentions in the MDSs, and so on.

To estimate the performance of a single MDS in the controlled environment, we have performed a metadata-intensive benchmark on CephFS where each client repeatedly creates empty files in its own directory (the detailed experimental setup is described in Section IV). Figure 1 shows that the aggregated throughput does not scale well when the number of clients varies from 1 to 512. In particular, we notice that the scalability of a dedicated metadata server is far from linear scalability and the throughput collapses as the number of clients is increased beyond 4.

[1]ArkFS does not support the atomicity of I/O operation that cross object boundaries. For example, if there are concurrent writes across two objects, the writes may be applied to each object in a different order. We note that this is also the case for CephFS. Nevertheless, as our target workload is the archiving workload which involves very few writes to the same file, we believe this will not be a significant problem.

ArkFS aims at solving this metadata scalability problem using the characteristics of the target environment where a large amount of data is moved between the burst buffer layer and ArkFS. Because ArkFS acts as a cold storage tier, data movement between the burst buffer layer and ArkFS is usually initiated by administrators or background daemons. In this case, we can schedule the data movement in a way that minimizes metadata conflicts. More specifically, ArkFS delegates the metadata management of a directory to the client that is accessing the directory without deploying a dedicated metadata server. Hence, the performance of ArkFS is maximized when the data movement jobs work on non-overlapping directories. In the same scenario shown in Figure 1, each client will create files locally and the generated metadata is flushed to the underlying object storage system directly without any communication with other clients during file system operations.

III. ArkFS Architecture

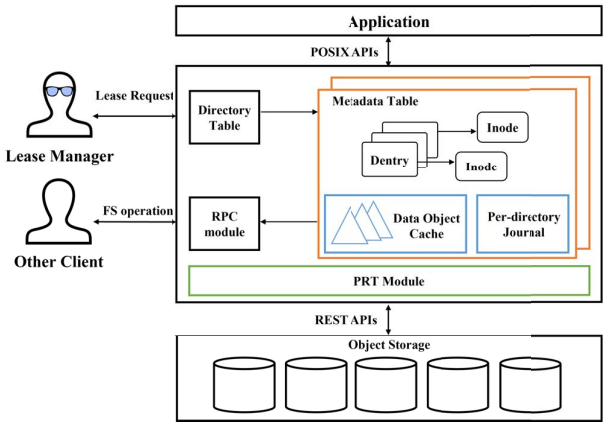


Fig. 2: **The internal design of ArkFS**.

A. Overall Architecture of ArkFS

We propose ArkFS which offers a near-POSIX interface with a client-driven metadata service on top of the object storage. ArkFS is designed to perform archiving jobs well in the controlled environment. To address the scalability problem in metadata handling which many existing DFSs suffer from, ArkFS takes advantage of the fact that most of the archiving jobs in the controlled environment are executed independently on different directories. Hence, ArkFS actively involves each client in the metadata management and holds them responsible for updating the metadata of the directory they are currently working on. However, even if we assume the controlled environment, it is quite possible for clients to work in the same directory in which we need coordination among them. Thus, ArkFS uses a lease mechanism [23] to delegate a right for managing and modifying specific metadata during the predefined period under the control of the *lease manager*.

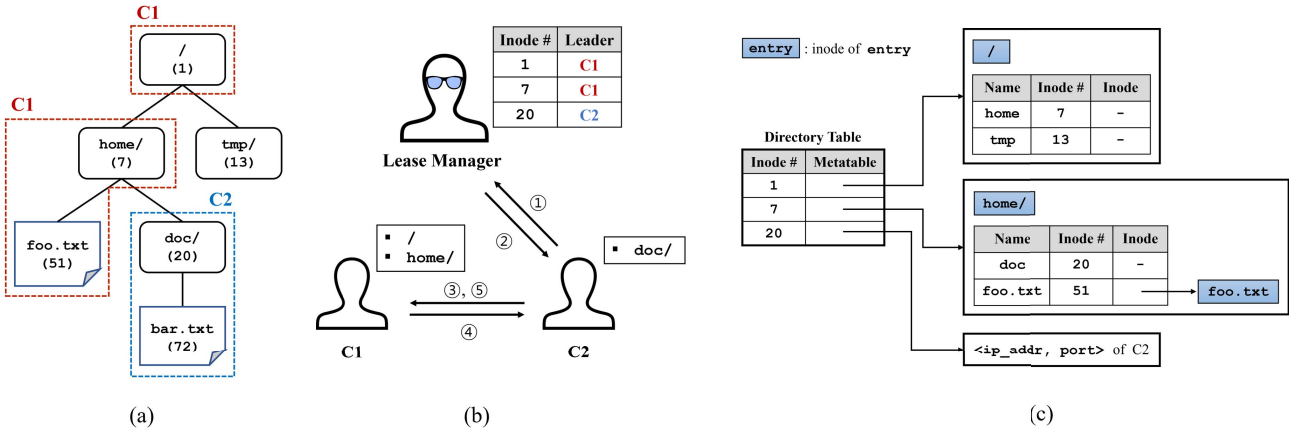
Figure 2 shows the internal architecture of ArkFS and its interactions with the lease manager and other clients. Let us suppose an application (e.g., an administrator process in the background) calls a `CREATE` operation. First, the ArkFS client tries to acquire a lease of the parent directory which contains the target file from the lease manager. When the client succeeds in acquiring the lease, it constructs a data structure called a *metadata table*. This metadata table is built on a per-directory basis and it contains metadata related to that particular directory. The metadata table allows the client to perform metadata operations in the local memory without any communication with remote components. Then, the client commits modifications including a newly created inode and a directory entry (*dentry*, for short) in the *per-directory journal*, which is in charge of storing all metadata modifications made in the corresponding directory. Finally, all POSIX block I/Os passed from applications or ArkFS internal components are translated to the REST object I/Os through our POSIX-REST Translator (*PRT*) module. We explain each ArkFS component in more detail in the following subsections.

B. Lease Management

ArkFS adopts a directory-based lease mechanism to manage rights to modify the file system metadata. ArkFS deploys a lease manager in the cluster and it issues a lease with a period of 5 seconds by default to a client before the client constructs the per-directory metadata table. If an application on a client attempts to access a directory, the client makes a lease request which contains network information of the client itself including IP address, port number, and the inode number of the requested directory. If nobody has the lease, the lease manager records the `<ip_addr, port>` pair with the inode number to redirect further accesses from other clients who want to get a lease for the directory.

The lease mechanism works in a first-come, first-served manner; if there are multiple lease requests, the lease is issued to the first requester. We call the client who succeeds in acquiring the lease a *directory leader* of that directory. Only the directory leader is allowed to modify metadata related to the corresponding directory and to manage I/Os for child files' data in the directory during the lease period. Note that each client may become the leader of multiple directories. In our target environment, working directories of multiple clients rarely overlap. However, we still need to coordinate accesses when multiple clients attempt to obtain a lease for the same directory. In this case, the rest of the clients who failed to get a lease should send their requests to the directory leader so that the directory leader can perform the requested operations on behalf of the other clients.

Figure 3(a) shows an example file system hierarchy where the dotted lines indicate the lease boundaries. Figure 3(b) illustrates the lease management process when the client *C2* tries to get a lease which is already held by another client *C1*. Let us assume the client *C1* already has the leases for the directories `/` (root directory) and `/home`. Now consider the situation where the client *C2* wants to create a text file named `baz.txt` in `/home`; ① First, *C2* sends a lease request for `/` to the lease manager. ② As `/` is being managed by *C1*, the manager rejects the request and passes the IP address of *C1*



(a) (b) (c)

Fig. 3: **An example of Client-driven Metadata Service** (a) the file system hierarchy where the number in parenthesis indicates the inode number. (b) an example scenario when *C2* tries to obtain a lease currently held by *C1*. (c) a set of *C1*'s metatables; the first two are local metatables and the last points to a remote metatable in *C2*.

to *C2* instead. ③ Now *C2* sends a lookup request for `/` to *C1*. ④ *C1* returns the result of the lookup request to *C2* and *C2* repeat it to access `/home`. ⑤ Finally, *C2* sends a `CREATE` operation to *C1* and *C1* performs the operation on behalf of *C2*.

Whenever a lease is acquired, the leader has to load the metadata from object storage to construct the per-directory metadata table. Even if the client was the leader of the directory previously, it has to reload the metadata because the metadata in memory might be out-of-date. However, if the lease is re-acquired by the same leader as soon as it is expired, the leader need not reload the metadata as the previous metadata table is guaranteed to be up-to-date. To optimize such cases, the lease manager keeps information of the previous leader for each directory and supports a lease extension. If a leader succeeds in extending a lease by acquiring it again, the leader can continue to work over the corresponding directory. As in our target scenario, less directory sharing removes a burden of the metadata flush because there would be few lease conflicts and all the metadata updates may already be reflected when the lease owner changes.

In the current implementation, ArkFS uses a single lease manager to control leases among clients. Acquiring/extending a lease is a very lightweight operation and we did not notice any performance degradation due to the use of a single lease manager during our experiments. Even when clients compete for access to shared directories, the role of the lease manager remains relatively unchanged, with the exception of needing to provide the IP of the leader to non-leader clients. However, a single lease manager may become a performance bottleneck in certain situations and it would be beneficial to implement distributed coordination using a cluster of lease managers. We leave this as future work.

C. Per-directory Metadata Table

ArkFS does not deploy a centralized metadata server, but clients voluntarily perform metadata management. This client-driven metadata service is made possible due to the per-directory metadata table (*metatable*, for short). When a client accesses a directory, the client tries to get a lease of that directory. If the client succeeds in getting the lease, it now has the right to build the metatable. First, the client reads the inode of the directory and checks whether the client has a right to access the directory. If not, the client releases the lease and returns a permission error. If the access permission is granted, the client loads several metadata from object storage (such as dentries and inodes of the child files, etc.) and constructs the metatable. After pulling all the metadata from object storage, the client becomes a directory leader of the particular directory who should ensure the integrity of the metadata during the lease period. Each client might be a leader of multiple working directories and the associated metadata are placed in local memory in the form of metatables. Consequently, all the metadata operations including the path-name resolution and permission checking can be done locally because it does not need to access the metadata server as is done in other DFSs.

Figure 3 demonstrates how ArkFS's client-driven metadata service is performed by two clients *C1* and *C2*. *C1* has already succeeded in acquiring leases of the root directory and `/home`. The corresponding metatables are depicted in Figure 3(c). We can see that there are two metatables for the directories `/` and `/home`, and each metatable has the complete information consisting of the directory inode, dentries, and the inodes of its child files. If *C1* wants to perform some metadata operations for the file `/home/foo.txt`, *C1* can perform them as local operations without issuing any remote requests. However, it is possible that a single client may not be the leader of all the working directories because some other clients may already have acquired leases for certain directories. Since the leader of one directory is responsible for all of the metadata operations on that directory, clients who are not a leader should send requests to the leader to perform any file system operation, as mentioned in III-B.

In Figure 3(c), we can see that the metatable entry whose

key (20) is the inode number of `/home/doc` contains the network information of *C2*. We call it a remote metatable and it is used as a pointer to the remote leader. If *C1* tries to perform metadata operations on `/home/doc/bar.txt`, *C1* should send requests to *C2*, the leader of the directory `/home/doc`, through the RPCs (Remote Procedure Calls) with the information in the remote table. If *C1* does not have a permission to access `/home/doc/bar.txt`, *C2* will return a permission error, otherwise *C2* will perform the requested operation on behalf of *C1*. As a result, this client-driven metadata service may partition and distribute the file system hierarchy according to each client's working directory set and ArkFS delegates the responsibility of ensuring POSIX consistency semantics to those clients.

In our target environment where clients rarely share their working directories, the client-driven metadata service will achieve high performance while guaranteeing strong metadata consistency since most of the operations can be performed locally with a metatable. However, there is an issue that may harm the scalability of ArkFS even when only a few clients exist. During `OPEN`, ArkFS should check the permission of each directory along the path from the root to the target directory in order to meet the POSIX consistency and access control model. In this case, it could be a great burden for the client who is the leader of the root directory because it would receive massive permission checking requests for the root directory. Not only the leader of the root directory, but the leaders of all the directories near the root have the same problem.

To alleviate this *near-root hotspot* problem, ArkFS offers a special permission caching mode which relaxes the POSIX consistency semantics of access control lists. This mode can be selected at the user's discretion. In the permission caching mode, when a client first accesses the remote directory, the permission information of that directory is cached until its lease expires. Hence, the lease expiration becomes a synchronization point and the modification of the directory permission during the lease period would be eventually visible to other clients after that point. With the permission caching, clients can handle the path-name resolution by themselves and the leaders of near-root directories can concentrate on their jobs.

D. Data Object Caching

ArkFS has its own user-level data object cache that basically serves the same functionality as the page cache in the kernel. The number of cache entries and the size of each entry are configurable parameters. By default, the cache entry size is set to 2MB. This large cache entry size might cause internal fragmentation that is critical to cache performance, but it would not be a problem considering that our target scenario is data archiving whose I/O patterns are mostly sequential.

Internally, the radix tree is used to index cached data objects. Due to the large cache entry size, it is very likely to have a shallow depth allowing for faster lookups. ArkFS's object cache works in a write-back manner and also adopts a read-ahead mechanism to improve the performance of sequential read workload. Each file has a read-ahead window and the file data belonging to the window is asynchronously read in advance. If the data for the file is repeatedly read in sequence, the window size will be increased up to the predefined maximum read-ahead size (8MB by default that is the same as in CephFS). To optimize `READ` operations, the window size is set to the maximum size immediately if `READ` starts from the very beginning of the file, expecting that the file will be read sequentially.

Although file sharing is not common under the controlled environment, ArkFS should avoid reading stale data from the client's data object cache. For this, ArkFS uses a read and write lease mechanism [16] that allows multiple shared read or an exclusive write of cache entries for a limited period. Unlike the lease of metatable, read/write leases are issued by the leader of the parent directory. In other words, each leader has a duty of child files' read/write lease management. Initially, all the clients are issued read leases for the target file from the leader of the parent directory when `OPEN` or `CREATE` is called. Because `READ` can be performed concurrently by multiple clients, each client can cache data objects with the read lease even though they are not the leader. When `WRITE` operation is called for the first time, however, the read lease may be upgraded to the write lease if there are no other clients who have read/write leases at that time, and the client can perform `WRITE` upon data object cache. If there are other clients who have read leases, the leader broadcasts cache flushing requests to prevent stale cache entries on other clients' object cache and lets the clients perform I/O operations directly on object storage. This read/write lease mechanism works effectively in the situation where file sharing is less common and it is suitable for the archiving scenario.

E. Crash Consistency with Per-directory Journaling

Because all the clients engage in metadata management and they are vulnerable to sudden failure, crash consistency is one of our important considerations to guarantee a consistent file system state. To provide a crash consistency, ArkFS leverages the journaling approach that is being used in many file systems and DBMSes. Generally, most journaling file systems keep a single journal area. However, previous research indicates that the single journal area could be a performance bottleneck due to serialized journal writings. To avoid such a bottleneck, ArkFS has one journal for each directory instead of one global journal area. We call it *per-directory journaling*.

The per-directory journal is created when a leader is building the metatable and all journal entries of the directory and its child files are committed to the per-directory journal. For example, if the modification time of a child file is renewed, the updated file inode will be written in the journal of the parent directory. Due to the per-directory journal, journaling operations can be run in parallel if the jobs are executed on different directories. In order to improve the journaling performance further, ArkFS supports compound transactions with multiple commit and checkpoint threads, buffering journal entries in an in-memory transaction for 1 second. When the

file system metadata is modified, ArkFS inserts the changes into the running transaction for the directory. Commit threads periodically turn the running transactions into committing transactions and write them to the journals. Once those transactions are journaled, the corresponding checkpoint threads are woken up and checkpoint the transactions to the original objects. Finally, the checkpoint threads invalidate the formerly-written transactions in the journal. Per-directory journals in one client are statically mapped to the corresponding commit and checkpoint threads depending on the directory inode numbers.

The obvious advantage of the per-directory journaling is that multiple journals allow parallel commits and therefore ArkFS can cope with bursty archiving workloads. However, operations that involve two directories should commit a journal entry in two different journals atomically. For example, during a `RENAME` operation where the source and destination paths are different, journal entries will be committed to the journals of both the source and destination directories. ArkFS adopts two-phase commit protocol [18] to address atomic commit across journals.

Only the leader of a directory can access its journal, so it is crucial that the journal should be treated carefully when the leader is changed. The leader of a directory should do its best to synchronize all the updates in memory before the lease is expired. If there is not enough time to do so, the leader tries to extend the lease. If the leader fails to extend the lease and the lease is eventually expired, the lease manager coordinates synchronizing the updates between the previous leader and the new leader. After that, the new leader checks whether the journal has any valid transactions. If there are valid entries remaining in the journal, it means there was a crash or a power loss and the new leader should do a recovery to keep the file system consistent. We examine two possible failure scenarios and their recovery processes in ArkFS.

1) ***Client Failure****:* If the new leader of a directory finds valid transactions in the per-directory journal, it means the predecessor has crashed. In this case, the new leader should recover the directory and its child files by scanning transactions in the journal and bringing them up-to-date. During recovery, the lease manager prevents other clients from acquiring the lease of the crashed directory and makes them wait for recovery to complete. Also, the manager waits at least the lease period which is pre-defined by the system to ensure that read/write leases issued by previous leader expires. At the end of the recovery, the manager renews the lease on the leader who performed the recovery. During this recovery process, the other clients who are not related to the crashed directory can still perform their jobs without any need to stop and restart.

2) ***Lease Manager Failure****:* If the lease manager has crashed, the system can be restored simply by restarting the lease manager. Once restarted, the lease manager waits for a pre-defined lease period to avoid the situation where two clients have the leases for the same directory simultaneously. During the manager shutdown, any client who has the lease can continue its work for its own directory until the lease is expired. However, the client must wait for the lease manager to be launched again to acquire a new lease of the directory which has not been accessed recently.

F. PRT Modules

The PRT (POSIX-REST Translator) module is the layer that is responsible for the translation from POSIX block I/O operations issued by applications or ArkFS components to REST object operations. ArkFS can support any kind of object storage backend by registering the corresponding REST APIs in the PRT module. The PRT module also defines specifications for how file system-related information is stored in the key-value pair. ArkFS uses 128-bit UUID (Universally Unique Identifier) for its inode number and constructs the key of each object by concatenating pre-defined prefix and the inode number. A pre-defined prefix for metadata would be one of `i(INODE)`, `e(DENTRY)` or `j(JOURNAL)`. The PRT module divides the file data into multiple objects if the file size exceeds the maximum object size defined by the object storage. To store file data as an object, its key is constructed by combining the prefix `d(DATA)` and the index value of the data.

IV. Evaluation

A. Experimental Setup

TABLE I
System configurations of public cloud cluster node

Instance	c5a.8xlarge	c5n.9xlarge
vCPU	32	36
Memory	DDR4 64GB	DDR4 96GB
Network	10Gbits	50Gbits
Disk	AWS EBS 32GB	AWS EBS 128GB x 4
Operating System	Ubuntu 20.04.3 LTS	
Kernel Version	5.15.0-1020-aws	

To demonstrate the performance and capabilities of ArkFS for the HPC storage system in the controlled environment, we build a cluster with AWS EC2 instances [2]. We use `c5n.9xlarge` for storage nodes. For client nodes, we use `c5a.8xlarge` nodes for scalability test and `c5n.9xlarge` nodes for the other experiments. The cluster consists of 16 storage nodes and the number of client nodes varies from 1 to 64. Table I shows the detailed configurations of AWS instances we use.

We choose four file systems to compare with ArkFS: CephFS [38], MarFS [26], S3FS [13] and goofys [6]. We construct Ceph RADOS storage v16.2.10 on 64 OSDs and deploy CephFS with two different mount options: FUSE mount (CephFS-F) and kernel mount (CephFS-K). The number of active MDSs has been varied depending on the experiment, but unless otherwise explicitly stated, we use 1 MDS for CephFS. We also deploy MarFS v1.12 on the same AWS cluster as Ceph RADOS, which is composed of two dedicated metadata nodes based on IBM SpectrumScale [8] v5.1.5 and 14 data

nodes with ZFS [37] v2.1.4. According to the existing studies [19], [20], it is recommended to use the *pftool* which is a parallel metadata/data operation utility using MPI and MPI-IO. However, the parallel copy or parallel list functions provided by the *pftool* were not directly compatible with the benchmark tools that we use and its open-source version [10] did not work in our environment, so we use *interactive interface* [20] that uses the FUSE mount of MarFS to measure its metadata operation performance. Two S3-based file systems, namely S3FS (v1.91) and goofys (v0.24.0), are used to compare the file I/O performance on top of the S3 object storage[2].

We use the FUSE framework [33] v3.9.0 to implement ArkFS and the gRPC framework [7] v1.46.3 for the network communication among clients. For all experiments, ArkFS is deployed on the same object storage as its competitors. For example, ArkFS is deployed on RADOS and S3 for the comparisons with CephFS and S3FS, respectively. The lease manager is deployed on one of the client nodes.

B. Micro-benchmarks

We conduct several micro-benchmarks to evaluate the overall performance of ArkFS. First, we use the `mdtest` benchmark [11] to measure the throughput of metadata operations. We perform the `mdtest` using two configurations defined by IO500 [27]: `mdtest-easy` and `mdtest-hard`.

`mdtest-easy` runs in three phases (`CREATE`, `STAT`, and `DELETE`) and measures the performance of metadata operations with empty files. Therefore, `mdtest-easy` excludes the effect of data I/O operations. In `mdtest-easy`, files are generated only in the leaf node of the directory tree hierarchy, and each process operates in its own leaf directory.

Unlike `mdtest-easy`, `mdtest-hard` operates in four phases (`WRITE`, `STAT`, `READ`, and `DELETE`) which include small-sized data I/O operations. We use 3901-byte-sized files for `mdtest-hard` that is the default value in IO500. In addition to the creation of files in leaf directories, the `mdtest-hard` benchmark also involves the distribution of files across multiple directories. Furthermore, the client processes of `mdtest-hard` conduct file operations on an arbitrary directory, simulating the usage in a shared directory environment.

Both `mdtest-easy` and `mdtest-hard` are performed with 1 million files using 16 processes. We call `fsync()` after each phase, causing all modifications to be flushed to the underlying storage.

mdtest-easy. Figure 4 shows the throughput results for three phases of `mdtest-easy`. Throughout all the phases, ArkFS shows a significant performance improvement over other file systems. The user-kernel context switching overhead caused by FUSE and the network round-trip overhead between

[2]We did not compare ArkFS against DAOS [25] because DAOS is designed and optimized for massively distributed systems with non-volatile memories. Since the environment we are targeting in this paper is the cold storage tier equipped with cheap and large-capacity storage devices such as hard disks, the direct comparison between ArkFS and DAOS seems inappropriate.

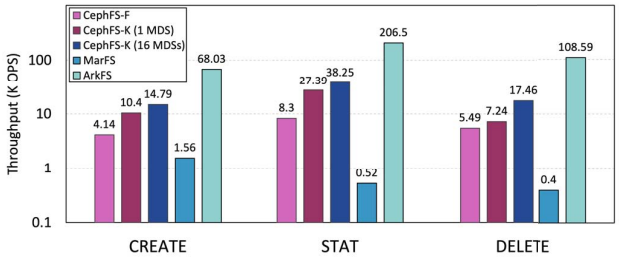


Fig. 4: **Throughput of mdtest-easy**. Throughput of metadata operations with empty files.

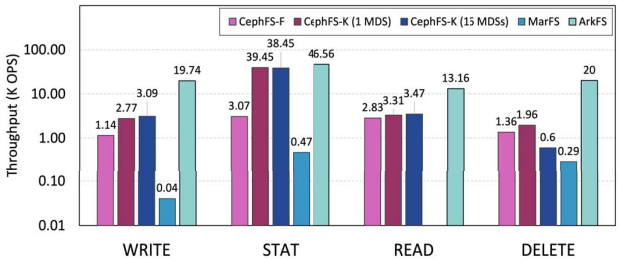


Fig. 5: **Throughput of mdtest-hard**. Throughput of metadata operations with small-sized files.

clients and metadata servers prevent MarFS and CephFS-F from achieving high throughput. CephFS-K does not have the FUSE overhead because it is an in-kernel file system, so its performance is better than that of MarFS or CephFS-F. Besides, CephFS-K (16 MDSs) demonstrates at most 2.41x better performance than CephFS-K (1 MDS). However, CephFS-K still shows worse performance compared to ArkFS due to its overhead of accessing metadata servers to process metadata operations over the network.

On the contrary, ArkFS can handle metadata operations in the local memory with the per-directory metatable and metadata modifications are gathered in compound transactions until they are committed. As a result, ArkFS shows much higher throughput in the `mdtest-easy` benchmark.

mdtest-hard. Figure 5 compares the throughput of ArkFS with CephFS-F, CephFS-K with two MDS setups and MarFS in each phase of `mdtest-hard`.

In the `WRITE` phase, ArkFS achieves higher performance than its competitors. Unlike the `CREATE` phase of `mdtest-easy`, the `WRITE` phase of `mdtest-hard` involves 3901-byte write operations following file creation, and is conducted in a shared environment as each process of `mdtest-hard` accesses an arbitrary directory. As a result, the performance difference observed in this phase is somewhat reduced compared to the `mdtest-easy`. Nevertheless, ArkFS shows higher performance because the directory leaders, who have the metadata they need in the per-directory metadata table, are able to perform metadata operations locally without reaching out to the remote metadata server and small file writes are efficiently processed with the aid of the data object cache.

For the `STAT` phase, we can observe that the performance

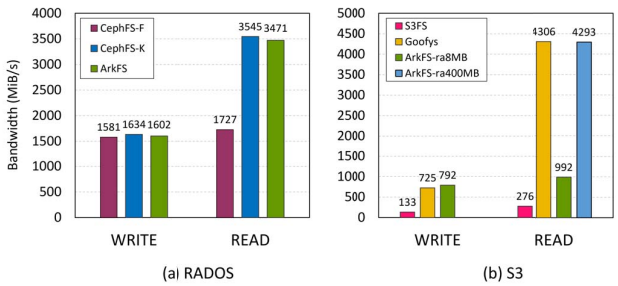


Fig. 6: **Large File I/O Bandwidth**. Comparison of the `WRITE`/`READ` bandwidth of various file systems deployed on RADOS (a) and S3 object storage (b).

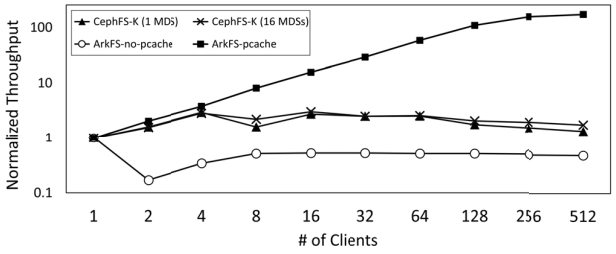


Fig. 7: **Scalability Test**. Massive file creations are performed while varying the number of clients. X-axis is the number of clients and Y-axis is the normalized throughput in log scale.

gap between ArkFS and CephFS-K is smaller compared to other phases. This is due to the FUSE's inherent limitation; for the LOOKUP operation, FUSE holds an exclusive kernel lock until the operation is completed by the user-space FUSE daemon. In ArkFS, this prevents clients from accessing metadata concurrently even though they can be handled simultaneously. Since the `STAT` phase of `mdtest-hard` incurs more lock contentions than that of `mdtest-easy`, the performance improvement of ArkFS decreases compared to CephFS-K. In spite of that, ArkFS still shows better performance than others because ArkFS eliminates the network overhead when processing metadata operations.

In the `READ` phase, ArkFS shows at most 4.65x higher performance than other file systems. Note that MarFS returns errors when we perform this phase in our environment.

Unlike `DELETE` phase of `mdtest-easy` which requires only metadata removals, `mdtest-hard` needs to delete the data generated during the `WRITE` phase. Despite this additional operations, ArkFS still shows better performance than others. Overall, ArkFS significantly outperforms both CephFS implementations and MarFS.

Unexpectedly, the results of `mdtest-hard` between CephFS-K (1 MDS) and CephFS-K (16 MDSs) demonstrate minimal performance differences, with the `DELETE` phase even showing a substantial performance decline. We surmise the overhead might come from the forwarded requests which incur additional network round trip among multiple MDSs and metadata migration caused by dynamic subtree partitioning [38] between MDSs. These overheads in the MDS cluster environment were addressed in recent studies [31], [36]. There are some configurable parameters and features (such as static directory pinning and Mantle [31] prototype) in Ceph that could potentially mitigate these overheads. However, optimizing the performance of the MDS cluster is highly dependent on the deployment environment and workloads, and it is beyond the scope of this paper.

fio. To demonstrate that ArkFS still shows good I/O performance for large file I/O, we perform the `fio` benchmark [4]. We run `fio` with 32 processes and each process writes and then reads a 32GB file using 128KB request size (total 1TB). At the end of the file writing, each `fio` process calls `fsync()` to ensure that all dirty data is written to the object storage and drops the cache entries of written files.

First, we compare the bandwidth of ArkFS with those of CephFS-F and CephFS-K on top of the RADOS object storage in Figure 6(a). We can see that three file systems demonstrate similar `WRITE` bandwidth. CephFS utilizes the kernel page cache in a write-back manner for dirty data and ArkFS also uses its data cache in the same way, so the result shows little differences in the `WRITE` performance.

In the case of `READ`, CephFS-K and ArkFS show almost the same performance because ArkFS also performs up to 8MB of read-ahead similar to CephFS, which is very effective for sequential reads. However, CephFS-F uses 128KB as a default max read-ahead size, so it shows lower `READ` bandwidth compared to other file systems.

We now compare ArkFS with S3FS and goofys running on top of AWS S3 in Figure 6(b). For brevity, we use ArkFS to refer to ArkFS-ra8MB that has the default read-ahead size of 8MB. For the `WRITE` performance, ArkFS shows 5.95x higher performance than S3FS. Because S3FS uses a disk as a data cache, data modifications are temporarily aggregated on disk and these files are written back to S3 when `fsync()` is called. Thus, this slow disk cache causes a substantial performance gap between ArkFS and S3FS. For such a reason, ArkFS also demonstrates 3.59x higher `READ` performance than S3FS. However, goofys shows much greater `READ` bandwidth than ArkFS. This is because goofys is extremely optimized for sequential reads; the max read-ahead size is set to 400MB in goofys which is 50x larger than the 8MB size in ArkFS. Thus, we repeat the same experiment by increasing the max read-ahead size of ArkFS to 400MB and add its result under the name of ArkFS-ra400MB in Figure 6(b). We can confirm that ArkFS-ra400MB shows the `READ` bandwidth similar to goofys.

The results demonstrate that ArkFS is capable of efficiently handling large file migration from/to the burst buffer layer by background administrator daemons, which is the intended use case of our system.

C. Scalability Test

To assess the scalability of ArkFS when performing metadata operations, we measure the throughput of file creation

TABLE II
Execution times of two archiving scenarios on each file system

	CephFS-F	CephFS-K	ArkFS	Speed-up
`Archiving` (sec)	2016.86	450.279	297.635	6.78x / 1.51x
`Unarchiving` (sec)	1791.239	837.353	475.93	3.76x / 1.76x

with the `mdtest-easy` benchmark while varying the number of clients up to 512. When the number of clients exceeded the the number of client nodes, the experiment has been conducted by running two or more client processes equally on each node. Note that each client process works on its own directory.

Figure 7 depicts the normalized throughput of ArkFS (without and with permission caching mode) and CephFS-K configured with 1 or 16 MDSs. In the case of CephFS-K with 1 MDS, a massive amount of file creation requests are directed to the single metadata server, which severely limits the overall scalability. Interestingly, we observe only a limited improvement when we increase the number of MDSs to 16 in CephFS-K. The CephFS-K (16 MDSs) just shows at most 3.24x improvement compared to the CephFS-K (1 MDS) for more than 64 clients. As we mentioned in the previous section, when used with multiple MDSs, CephFS attempts to dynamically distribute request loads based on the popularity of the metadata [38], but this results in additional communications among servers and periodic transfers of metadata.

To examine the scalability of ArkFS. we test two types of ArkFS: one without the permission cache mode (ArkFS-no-pcache) and the other with the permission cache mode (ArkFS-pcache). First, we can see that ArkFS-no-pcache demonstrates a drastic performance degradation when the number of clients is increased to 2. What makes the performance worse is the way the FUSE driver checks the existence of a directory [33]. When an application process issues a metadata operation, the FUSE driver sends multiple LOOKUP operations to the user-level FUSE daemon along the target file's pathname. For example, if an application calls `CREATE(/home/foo.txt)`, it incurs three `LOOKUP` requests, (for `/`, `home`, and `foo.txt`) and a single `CREATE` request to ArkFS, and ArkFS performs path traversal on each request. These additional requests cause many network communications to check the permission information which would slow down the leaders of near-root directories. This kind of FUSE behavior and *near-root hotspot* problem harm the scalability of ArkFS, so ArkFS-no-pcache shows a severe performance drop and less scalability as the number of clients is increased.

As mentioned in section III-C, we adopt the permission caching mode to alleviate this problem and we evaluate how much it affects the scalability of ArkFS. In Figure 7, ArkFS-pcache presents near-linear throughput scalability while the number of clients is increased up to 512. The permission caching mode of ArkFS helps reduce the burden of excessive permission checking requests in the near-root directory leaders by allowing each client to cache the permission information of the parent directories during its lease period. In addition, the per-directory metadata table and the per-directory journal make the ArkFS client perform file creation and its journal commit without any disruptions from other clients.

D. Archiving Workload

To demonstrate the usefulness of ArkFS for archiving workloads, we construct two synthetic scenarios using *tar* [5] which simulate the data migration between the burst buffer and campaign storage. These scenarios were then executed on both CephFS and ArkFS. The scenarios that we assume are as follows:

1) ***Archiving****:* When the dataset has not been accessed for a specified period of time, it is kept in the form of a tar file and moved to campaign storage by the administrator daemon or data lifecycle management tools which run in the background. Next, the dataset is extracted from the tar file and categorized by its date or its data type, etc.

2) ***Unarchiving****:* From the archived dataset in the campaign storage, users retrieve the data that will be used soon in the form of a tar file and move it back to the burst buffer.

We use the MS-COCO [12] image dataset which is commonly used for object detection and store it in the AWS Elastic Block Store (EBS) [1] with a sequential bandwidth of 1GB/s. The MS-COCO dataset has 41K images with sizes ranging from tens to hundreds of KB and an aggregated size of `7GB`. We conduct the workload using 32 concurrent processes and each process handles a single MS-COCO dataset (`224GB` in total).

Table II illustrates the elapsed time of each scenario and demonstrates how far ArkFS surpasses the performance of the two mounted types of CephFS. In the table, *Speed-up* indicates the performance improvement of ArkFS over CephFS-F and CephFS-K. We can see that ArkFS presents 6.78x and 3.76x faster execution time than CephFS-F in both workloads. In addition, ArkFS demonstrates 1.51x and 1.76x faster execution time than CephFS-K. However, the result compared to CephFS-K shows lower improvement than that of other benchmarks. This is due to the limited bandwidth of AWS EBS where the dataset `READ`/`WRITE` time from the external storage takes up a nontrivial portion of the total elapsed time. We expect that if we were to use higher-performance external storage, the performance improvement of ArkFS would be even more noticeable.

Nevertheless, the results indicate that ArkFS is capable of effectively handling archiving workloads within the controlled environment. While CephFS suffers from network round-trip overheads, ArkFS can perform metadata operations locally

because each client is likely to be a leader of the parent directory of the image files. Furthermore, the per-directory metadata management scheme in ArkFS aligns well with the functionality of *tar* since this tool collects or extracts files from/to a single directory at a time.

V. Conclusion

We present ArkFS which offers a near-POSIX compatibility with a client-driven metadata service on top of the object storage. ArkFS manages metadata on a per-directory basis which allows clients to serve operations in parallel in the controlled environment. ArkFS also guarantees the POSIX consistency semantics among multiple clients by leveraging a lease mechanism. Moreover, ArkFS can provide rich POSIX features on any kind of distributed object storage with the assistance of the PRT module that translates incoming POSIX operations to REST operations. Our evaluation shows that ArkFS can provide high performance metadata operations with near-linear scalability and can effectively support the archiving role under the controlled HPC environment.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant (No. 2019R1A2C2089773) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-0-01363) funded by the Korea government (MSIT).

References

[1] Amazon Elastic Block Store. https://aws.amazon.com/ebs/.
[2] Amazon Elastic Compute Cloud. https://aws.amazon.com/ec2/.
[3] Amazon Simple Storage Service. https://aws.amazon.com/s3/.
[4] FIO benchmark. https://fio.readthedocs.io/en/latest/fio_doc.html.
[5] GNU Tar. https://www.gnu.org/software/tar/.
[6] goofys. https://github.com/kahing/goofys.
[7] gRPC. https://grpc.io/.
[8] IBM Spectrum Scale. https://www.ibm.com/products/spectrum-scale.
[9] IEEE Std 1003.1™-2017 (Revision of IEEE Std 1003.1-2008). https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.
[10] LANL pftool. https://github.com/pftool/pftool.
[11] MDTEST benchmark. https://github.com/LLNL/mdtest.
[12] MS-COCO. https://cocodataset.org/.
[13] S3FS-FUSE. https://github.com/s3fs-fuse/s3fs-fuse.
[14] SwiftFS. https://github.com/wizzard/SwiftFS.
[15] The Swift Virtual File System. https://github.com/ovh/svfs.
[16] Thomas E Anderson, Michael D Dahlin, Jeanna M Neefe, David A Patterson, Drew S Roselli, and Randolph Y Wang. Serverless network file systems. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 109–126, 1995.
[17] Joe Arnold. *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. " O'Reilly Media, Inc.", 2014.
[18] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.
[19] Peter Braam and Dave Bonnie. Campaign storage. In *Proceedings of the International Conference on Massive Storage Systems and Technology*, pages 1–8, 2017.
[20] Hsing-Bung Chen, Gary Grider, and David Richard Montoya. An early functional and performance experiment of the marfs hybrid storage ecosystem. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 59–66. IEEE, 2017.
[21] Jeffrey Dean. Evolution and future directions of large-scale storage and computation systems at google. 2010.
[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.
[23] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.
[24] Jan Heichler. An introduction to BeeGFS, 2014.
[25] Michael Hennecke. Daos: A scale-out high performance storage stack for storage class memory. *Supercomputing frontiers*, page 40, 2020.
[26] Jeffrey Thornton Inman, William Flynn Vining, Garrett Wilson Ransom, and Gary Alan Grider. Marfs, a near-posix interface to cloud objects. *; Login*, 42(LA-UR-16-28720; LA-UR-16-28952), 2017.
[27] Julian Kunkel, Gerald Fredrick Lofstead, and John Bent. The virtual institute for i/o and the io-500. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.
[28] Yubo Liu, Yutong Lu, Zhiguang Chen, and Ming Zhao. Pacon: Improving scalability and efficiency of metadata service through partial consistency. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 986–996. IEEE, 2020.
[29] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.
[30] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE, 2014.
[31] Michael A Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A Brandt, Sage A Weil, Greg Farnum, and Sam Fineberg. Mantle: a programmable metadata load balancer for the ceph file system. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.
[32] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. Someta: Scalable object-centric metadata management for high performance computing. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 359–369. IEEE, 2017.
[33] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017.
[34] Murali Vilayannur, Samuel Lang, Robert Ross, Ruth Klundt, Lee Ward, et al. Extending the posix i/o interface: a parallel file system perspective. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2008.
[35] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. Understanding lustre filesystem internals. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep*, 120, 2009.
[36] Yiduo Wang, Cheng Li, Xinyang Shao, Youxu Chen, Feng Yan, and Yinlong Xu. Lunule: an agile and judicious metadata load balancer for cephfs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2021.
[37] Scott Watanabe. *Solaris 10 ZFS Essentials*. Pearson Education, 2009.
[38] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.
[39] Brent Welch. POSIX IO extensions for HPC. 2005.
[40] Qing Zheng, Kai Ren, and Garth Gibson. BatchFS: Scaling the file system control plane with client-funded metadata servers. In *2014 9th Parallel Data Storage Workshop*, pages 1–6. IEEE, 2014.
[41] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. DeltaFS: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 1–6, 2015.