2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)

Evaluation of HPC Application I/O on Object Storage Systems

Jialin Liu[1], Quincey Koziol[1], Gregory F. Butler[1], Neil Fortner[2], Mohamad Chaarawi[3], Houjun Tang[1], Suren Byna[1], Glenn K. Lockwood[1], Ravi Cheema[1], Kristy A. Kallback-Rose[1], Damian Hazen[1], Prabhat[1]

[1] Lawrence Berkeley National Laboratory, [2] The HDF Group, [3] Intel Corporation

***Abstract*— POSIX-based parallel file systems provide strong consistency semantics, which many modern HPC applications do not need and do not want. Object store technologies avoid POSIX consistency and are designed to be extremely scalable, for use in cloud computing and similar commercial environments. In this work, we evaluate three object store systems: Intel DAOS, Ceph RADOS, and Openstack Swift, and evaluate them with three HPC applications: VPIC, H5Boss, and BDCATS. We have developed virtual object layer (VOL) plugins for HDF5 that can redirect the applications' HDF5 calls to the underlying object storage systems' APIs, with minimum application code change. Through our evaluation, we found that object stores have better scalability in many cases than POSIX file systems, but are not optimized for common HPC use cases, such as collective I/O. Understanding current object store I/O details and limitations will enable us to better design object stores for future HPC systems.**

I. Introduction

HPC storage architectures are becoming increasingly complex as we enter the exascale computing era. The storage hierarchy in upcoming HPC systems is expected to range from node-local storage in the form of storage class memory (SCM) and solid state disk (SSD), shared burst buffer layer with SSDs, parallel file systems with hard disk drives, and the continued presence of a tape archive. On the other hand, scientific applications using HPC systems are exacerbating the complexity even further with the increasingly massive amounts of data they are generating or consuming (or both). The amounts of data produced or analyzed by HPC applications from science simulations from high-energy physics [24], plasma physics [16], etc., and experimental facilities such as Advanced Light Source (ALS) [1], LCLS-II [9], and CryoEM [42] are expected to reach exabytes in the near future. A critical requirement in HPC systems with these large-scale applications is to be able to store and retrieve this scale of data efficiently.

Parallel file systems along with I/O middleware (MPI-IO) and high-level I/O libraries (HDF5, netCDF, etc.) have been serving data storage needs on HPC systems for decades. However, the POSIX and MPI I/O standards that are the basis for existing I/O libraries and parallel file system interfaces present fundamental challenges in the areas of scalable metadata operations, semantics-based data movement, performance tuning, asynchronous operations, and support for scalable consistency of distributed operations. Several efforts have proposed workarounds to full POSIX compliance and also extensions to the POSIX I/O interface to handle large-scale parallel I/O. For example, Vilayannur et al. [44] propose extensions to POSIX I/O for PVFS with the goal of providing hints about I/O patterns and performing lazy metadata operations. Parallel file systems, such as Lustre, GPFS, and Cray Data Virtualization Service (DVS) provide various workarounds [43]. Despite these patchwork solutions, moving beyond the POSIX I/O API is needed for HPC to achieve exascale I/O goals [13], [17], [12], [44].

'Object-based storage' [37], [30] is a generic term used to describe an abstract data container that consists of many byte-streams (or objects), each with related attributes. Attributes are stored and transferred with the objects and object-based storage can efficiently express quality-of-service, transparent performance optimizations, data sharing, and data security qualities that an application can exploit.

Several object-based storage solutions are either entering use or in development currently. Among them, Intel is developing DAOS (Distributed Asynchronous Object Storage), an open-source software-defined object store designed from the ground up for massively distributed nonvolatile memory (NVM) [22]. DAOS takes advantage of next generation NVM technology like Storage Class Memory (SCM) and NVM express (NVMe), while presenting a key-value storage interface and providing features such as transactional non-blocking I/O, advanced data protection with self-healing on top of commodity hardware, end-to-end data integrity, fine grained data control and elastic storage to optimize performance and cost. [4]. DAOS is a complete I/O architecture that aggregates SCM and NVMe storage distributed across the fabric into globally-accessible object address spaces, providing consistency, availability and resiliency guarantees without compromising performance.

RADOS [14], as part of Ceph [45], is another scalable and reliable object storage service for petabyte-scale storage clusters. The Ceph architecture can be neatly broken into two key layers. The first is RADOS, a reliable autonomic distributed object store, which provides an extremely scalable storage service for variably sized objects. The Ceph file system is built on top of that underlying abstraction: file data is striped over objects, and the MDS (metadata server) cluster provides distributed access to a POSIX file system namespace (directory hierarchy) that's ultimately backed by more objects. Until now, RADOS' only user has been Ceph.

OpenStack Swift [11], The OpenStack object store project, known as Swift, offers cloud storage software to store and retrieve large volumes of data with a simple API. It's built for scale and optimized for durability, availability, and concurrency across the entire data set. Swift is ideal for storing unstructured data that can grow without bound.

978-1-7281-0192-7/18/$31.00 ©2018 IEEE
DOI 10.1109/PDSW-DISCS.2018.00006

With the goal of evaluating object storage solutions on HPC systems with science use cases, we present an evaluation of Intel DAOS, Ceph RADOS, and OpenStack Swift. NERSC supports 7000 users for mission critical science on the center's flagship systems, Cori and Edison. Evaluating object store technologies with representative science applications is beneficial to the broader HPC community and sheds light on the next generation of HPC storage systems. We have configured a small testbed (4 clients, 2 gateways, and 4 storage nodes) at NERSC for this evaluation. To provide the software bridge between existing high-level HPC I/O interfaces and object stores, we have developed plugins for HDF5, a popular I/O library, and the selected object storage systems. HDF5 provides a Virtual Object Layer (VOL) feature that allows interception of HDF5 public API calls and redirection of the calls to object interfaces. We have developed VOL plugins for RADOS and Swift in this effort. We have also updated the existing prototype DAOS VOL plugin as part of this work.

The contributions of this work include:

1) a comprehensive evaluation of object store technologies with several HPC applications
2) three HDF5 VOL plugins were either developed for the first time (RADOS and Swift) or enhanced (DAOS) during this evaluation
3) a better understanding of various object store I/O internals and their potential in resolving the POSIX constraints.
4) lessons for future HPC object store design and optimization.

We first review the difference between object store and file system in Section II. In Section III, we describe the object store testbed setup. A brief discussion regarding object store's function test is included in Section IV. Three HDF5 Virtual Object Layer I/O plugins design and their evaluation with three HPC applications are discussed in Section V and Section VI separately. We cover a few related work in VII and conclude the work in Section VIII

II. Object Stores vs. File Systems

The vast majority of traditional HPC applications have long relied on file systems to store application input and output data. Today's file systems support the POSIX I/O interface, including functions such as `open()`, `close()`, `read()`, and `lseek()`, and implement simple assurances about the behavior of these functions. For example, the POSIX standard stipulates that data that is successfully written using the `write()` call must be immediately available to be read using the `read()` call.

While this semantic meaning of `read()` and `write()` is trivial at first glance, it introduces a tremendous amount of complexity on networked and parallel file systems where clients are not aware of which other clients may be modifying the file. As a result, many parallel file systems (including Lustre [21] and Spectrum Scale (formerly GPFS) [41]) choose to implement distributed locking schemes ensure this *POSIX consistency*, effectively trading I/O performance for strong consistency, by serializing I/O traffic [34]. Such serialization is the antithesis of scalable performance though, and some parallel file systems (including PVFS2 [25] and DataWarp [32]) have chosen to recover some of this performance by providing relaxed, "POSIX-like" behavior that does not strictly conform to these POSIX semantics.

The foundations of file systems in POSIX ultimately result in inescapable points of serialization in the kernels of networked file system clients and servers, though. In response to the fundamental scalability limitations of POSIX file systems, object stores have emerged as a dramatically more scalable storage paradigm that reject the notion of file-based I/O and instead provide a much simpler interface for interacting with data objects. The key features that distinguish object stores from file systems are:

1) **Object stores have a flat namespace and no prescribed metadata schema.** Objects simply have a globally unique object ID, and all other metadata is optional and implementation-specific. This contrasts sharply with file systems which feature hierarchies of directories and files, owners and groups, hierarchical permissions, and a fixed set of metadata which includes file name, create/modify/access times.
2) **Object store access occurs through stateless, atomic operations such as PUT, GET, and DELETE.** Unlike file-based I/O, there is no stateful file handle or a requirement that a file be opened before its data can be accessed. A user simply requests all of the data corresponding to an object ID, or puts an entire data object and receives an object ID in response.
3) **Objects are immutable.** Once an object is written via PUT, it can never be modified again. This, combined with the aforementioned atomicity of the PUT operation, obviates the need for any type of distributed locking mechanisms. If data can be retrieved by GET, it is guaranteed to be consistent and immutable.

These three features allow object stores to scale out beyond the capability of file systems by not attempting to provide the broad feature set dictated by POSIX-like file systems. The corollary, though, is that object stores can be much more difficult for users and applications to use; accessing data based on object IDs in a flat namespace is analogous to only using inodes to reference files, and the immutability of objects makes object stores unaccommodating of highly dynamic data such as source code or application input configurations. As a result, object stores have historically been limited to industries with both tremendous data volumes and significant write-once, read-many (WORM) workloads such as content distribution [19], [38]. Following this use case, the vast majority of existing object storage systems today have been designed with scalable capacity for WORM workloads as the principal optimization point.

Over the last five years, the ever-increasing demands for scalable performance and capacity from the HPC community, combined with the increased affordability of nonvolatile memory and solid-state storage, have started shifting the

trajectory of extreme-scale parallel storage systems. The latency penalties of file-based I/O and passing I/O requests to the operating system kernel are becoming the single largest bottlenecks in I/O performance to emerging nonvolatile storage technologies. As a result, a new type of object stores that are optimized for extremely high performance and low latency are now being developed as a wholesale alternative to the traditional parallel file system [29], [36].

Thus, there are two broad categories of object stores that have applications in high-performance computing:

- **Hot archive object stores** are those optimized for highly scalable capacity and manageability but modest bandwidth and latency. This architecture naturally fits the role currently fulfilled by tape-based archives and colder file storage [28], and countless open-source and proprietary implementations are available and in use in enterprise computing environments.
- **High-performance object stores** are those optimized specifically for the low latency and high bandwidth enabled by emerging nonvolatile memory technologies. This architecture is designed to overcome the performance limitations of parallel file systems and, as such, are intended to displace burst buffers and parallel file systems. DAOS [36] and Mero [29] are perhaps the two most widely known.

It is important to note that these two categories are extremes at opposite ends of a spectrum, and some object store implementations (such as Ceph [46]) fall somewhere between these two. In addition, these categories only serve to characterize the lowest-level architectural features of the object store. It is not uncommon to implement much more semantically rich interfaces, including POSIX-like file systems, on top of these more primitive foundations [33], [23], [28], [26].

In this work, we explore implementations of object stores that represent both hot archives and high-performance object stores to provide a broad description of how these technologies may be applied in future HPC storage systems.

III. Storage Testbeds

A. SWIFT and RADOS Testbeds at NERSC

Two object store testbeds were configured at NERSC: a GPFS/Swift testbed and a Ceph/RADOS testbed. The testbeds used identical hardware and are all running CentOS 7.2.1511 (GPFS/Swift) or 7.3.1611 (Ceph/RAIDOS). All tests were conducted from Cori's Haswell nodes (each with 32 cores per node, and 128GB RAM).

1) Testbed Configuration: The testbeds use identical storage configurations. Each testbed consists of eight disk arrays and four primary storage servers. Each testbed also has two additional servers performing specialized roles depending on the file system being tested. These additional servers are identical to the storage servers.

2) Network Connectivity: All servers are connected to an FDR InfiniBand fabric, allowing them to communicate with each other using both IP and RDMA. Each server is connected to the IB fabric switch by a single QDR link from a QDR HCA. Access to these servers from outside of the testbed environment is through gateway systems attached to both the testbed InfiniBand fabric and external Ethernet networks.

3) Storage Connectivity: The four servers in the Swift testbed are configured as two pairs of active-active failover partners. Each pair serves four dual attached disk arrays. Each member of a pair acts as the primary server for half of the storage and the secondary server for the other half of the storage and is capable of serving all of the the pair's storage in case of failure of the partner server.

The four servers in the Ceph/RADOS testbed are independent and are not configured with any failover capability. Each server is configured to serve dedicated storage that belongs exclusively to them.

The storage arrays are connected to the servers via 8 Gb/s Fibre Channel connections, with each server having 8 connections, one to each of the two controllers belonging to each of the 4 arrays it serves.

4) Servers: The servers have Supermicro X8DAH+ motherboards with two 12 core 2.8 GHz Intel X6500 cpus, 48 GB of 1333 MHz DDR 3 ECC memory, 2 quad port FC8 Qlogic Fibre Channel adapters, and two dual port Mellanox ConnectX-2 QDR InfiniBand HCAs.

5) Storage: The storage arrays are Nexsan E60 arrays having two active-active redundant controllers and 60 3TB disk drives. The disk drives are organized into six 8+2 Raid 6 LUNs of 24TB each. Each of the controllers has two FC8 connections and is primary for 3 of the LUNs and secondary for the other 3.

6) Testbed specifics - Swift Testbed: The Swift testbed is using GPFS 4.2.3.4 protocols. It consists of 4 NSD servers operating as two failover pairs, with each pair serving 24 LUNs for a total of 48 LUNs/NSDs. Each NSD server is responsible for being the primary server for 12 of the LUNs and the secondary for the other 12.

The testbed also has two CES protocol servers that provide high-availability access to the underlying GPFS filesystem as an object store using openstack-swift 2.7.2. The Swift testbed also provides swift-on-file access to the underlying GPFS filesystem.

The underlying GPFS filesystem has a capacity of 1.1PB. Both the native GPFS filesystem and its Swift interface are exported to external systems through the InfiniBand to Ethernet gateways.

7) Testbed specifics - Ceph/RADOS Testbed: The Ceph/RADOS testbed is running Ceph 10.2.10 (Jewel). It consists of 4 OSD servers and two Monitor nodes, one of which is also a RADOS Gateway for access using Swift.

The Ceph/RADOS testbed has the standard configuration of 4 storage servers, each of which operates independently with non-shared storage. They serve a total of 48 LUNs having a total capacity of 1.1PB. Each of the 48 LUNs is configured as an OSD.

The Ceph/RADOS object store is available within the testbed environment and on external systems using both the

Rados CLI and Swift access mechanisms.

B. Lustre File System at NERSC

The Lustre filesystem (v2.7.2.25) is provided by Cray, and is used at NERSC by both the Cori and Edison systems, as a shared global scratch space. There are 248 Object Storage Servers (OSS), with each managing one Object Storage Target (OST). InfiniBand connects five metadata servers and 248 OSSs to the LNET routers on the Cray XC system. The OSTs are configured with GridRAID, similar to RAID6. Each OST consists of 41 disks, and can deliver 240TB of capacity. The maximum aggregated bandwidth is above 740 GB/sec. The I/O stack on this production system has been highly optimized, not only by Cray, but also from decades of effort in the HPC community (e.g., MPI, and Lustre). Thus, we do not claim the comparison to the three object store testbeds in this study is fair in any sense, but rather is a demonstration of the pros/cons of using object store technologies for future HPC systems.

C. DAOS Testbed at Intel

DAOS is installed on the Boro cluster at Intel. The Boro cluster has 80 nodes, with each node having 128 GB memory, 72 CPUs, and CentOS Linux 7.4.1708. It uses Intel Infiniband AXX1FDRIBIOM Single Port FDR I/O modules with QSFP to connect the nodes. The client-server communication is based on Mercury, using the OFI and PSM2 network providers. In this study, we used tmpfs as the storage backend for evaluating DAOS, since SCM and NVMe backends were not available on the Boro cluster at the time of this study.

IV. Object Store Functional Test

Any adoption of an object-storage solution will require a transition period for users to adapt code and workflows that, to date, have relied upon POSIX semantics. The Spectrum Scale Object services implementation, based on OpenStack Swift, provides an *objectization* feature to convert files to objects automatically and therefore may be useful for those users who would like to experiment with an object-based solution while not leaving the traditional file-based world all at once. This feature is a subset of the larger Spectrum Scale Cluster Export Services (CES) and, when enabled, is available cluster-wide.

For *Unified* file and object access, an administrator creates a storage policy of this type and a Spectrum Scale fileset is created for the policy. The *objectizer* is a scheduled process which asynchronously creates an object version of new files. It runs, by default, on a 30 minute cycle, or alternatively an administrator can specify files to be immediately *objectized*. The *objectizer* process is not needed for files created via the object interface to be viewed via file access. It is worth noting that there are separate ACLs for object versus the file access, so user access via each method can be controlled independently. This could be useful, for example, in a use case where there is internally-facing file access and externally-facing object access. The path to file access is

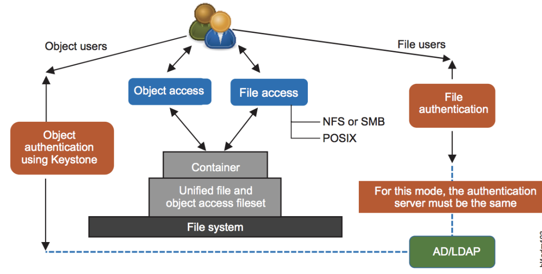

Fig. 1. Unified file and object access in IBM GPFS [7]

auto-generated based on the fileset name, the policy index number (same as region number), the keystone account ID.

We performed basic functional testing of *Unified* file and object access including Swift token creation for authentication, Swift upload of files, viewing these objects via file access, updating the metadata associated with the object as well as creation of files and confirmation of the object counterpart being created and accessible. All of this testing was successful and demonstrated that the system was operating correctly.

V. HDF5 Virtual Object Layer (VOL) Plugins

HDF5 provides a data model for storing array data in a directory-like structure. The data arrays themselves are called datasets and the directory objects are called groups. While the HDF5 library normally stores its data in a binary file using the native HDF5 file format, a new prototype feature called the Virtual Object Layer (VOL) allows HDF5 to interface with any arbitrary storage system, provided an appropriate plugin. A VOL plugin defines its own way of storing HDF5 objects that does not need to use the HDF5 file format or even a traditional file system. HDF5 API calls that would normally result in I/O to a file are instead forwarded to the VOL plugin, which executes the requested operation using the storage scheme it implements on top of the storage system it uses.

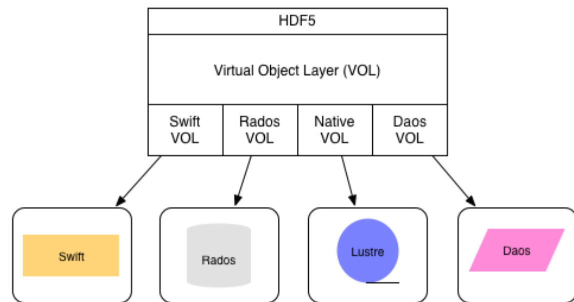

Fig. 2. Virtual Object Layer

This architecture, together with the object-based nature of HDF5 itself, make a VOL plugin a natural fit for running HDF5 applications on an object store. While native HDF5 translates object-based operations into a serial file format, a

VOL plugin can forward these operations to similar object-based operations in the object store API. This provides the advantages of an object store, where the storage system can make more intelligent decisions about data location and prefetching, without needing to re-code an application.

As shown in the Figure 2, we have developed three VOL plugins that interface with HDF5 API. For existing HPC applications, the only application code change required is to enable the VOL plugin, e.g., in order to use the Rados object store, the following two lines are added to the application:

```
H5VL_rados_init();
H5Pset_fapl_rados();
```

Following that, the application's HDF5 API calls will be automatically translated into the RADOS object store's API by the RADOS VOL plugin.

A. DAOS VOL Plugin

The DAOS VOL plugin currently supports most HDF5 features, including groups, datasets, named datatypes, and attributes. I/O support includes partial I/O, chunked datasets, datatype conversion, and variable-length datatypes, along with a prototype implementation of asynchronous I/O.

1) Interface: While the concept of versioned epochs is a fundamental part of the DAOS API, allowing multiple versions of containers (i.e. HDF5 files) to be stored and later retrieved, the most recent version of the DAOS plugin hides this from the application. All operations are performed on a single epoch, which is committed on a call to `H5Fflush()` or `H5Fclose()`. Subsequent operations on that file then use the next epoch. The plugin also provides additional API functions to enable storing and loading snapshots of the file, a feature made possible by the underlying use of DAOS epochs.

One major difference from native HDF5 that remains is that, while all metadata write operations in native HDF5 must be collective, in the DAOS plugin they are by default independent. This aligns the plugin with the highly independent design of DAOS, and allows applications to leverage DAOS' capability for highly independent access to the container. We believe that giving applications the option to create file objects independently will allow developers more flexibility in creating applications and could be a major reason for them to adopt HDF5/DAOS as opposed to native HDF5. To ease application porting, there is also an option to use the more traditional collective metadata access.

2) Mapping: The central paradigm for mapping HDF5 to DAOS is that HDF5 files are stored as DAOS containers, and HDF5 objects (groups, datasets, named datatypes, and maps) are stored as DAOS objects, using a 1:1 mapping. All metadata and raw data associated with an HDF5 object is then stored as entries in the key-value store associated with that DAOS object.

Since DAOS containers are identified by a UUID, the plugin uses a hashing function to generate a UUID from the file name (duplicated hash values are not handled in this prototype plugin). Each container for an HDF5 file contains at least two DAOS objects: a global metadata object and the HDF5 root group. The global metadata object stores metadata that describes properties that apply to the entire container. Currently this is only the maximum object ID used by the plugin. The root group is always the start of the HDF5 group structure, and uses the same format as all other groups.

All data and metadata for each HDF5 object is stored in a single DAOS object's key-value store. DAOS keys are actually represented as two parts: a *dkey* and an *akey*. The *dkey* determines the locality of the data – all data with the same *dkey* is co-located, while the *akey* has no such effect.

For all HDF5 objects, constant metadata is serialized and stored in records with the same *dkey*, and all HDF5 attributes are stored in another *dkey*. For HDF5 groups, each link is stored under a separate *dkey*, with the value being the DAOS object ID of the link target for hard links, or the target's path for soft links. For HDF5 datasets, each data chunk is stored using a separate *dkey*, with the value being the data buffer for that chunk.

3) Object IDs: DAOS objects are referenced through the DAOS API by a 128-bit object ID. Only the lower 64 bits are currently used by HDF5. The lowest 62 bits are simply set in increasing order, starting from 1, which is always the root group (OID 0 is reserved for the global metadata object). The remaining 2 bits are used to encode the HDF5 object type. This removes the need to store the object type in the key-value store for the object itself, and allows the plugin to determine the object type and therefore the routines used to access it without needing to query DAOS, reducing the number of metadata operation server requests. The lower 64 bits (including the encoded object type) are considered the address for the purposes of the HDF5 API, and is what is returned as `addr` from `H5Oget_info()` and what is accepted for `H5Oopen_by_addr()`.

Object ID allocation is currently not handled correctly when done independently by multiple processes. This is a temporary limitation, as we haven't incorporated the latest DAOS unique object ID allocator. This means that the object ID space should be divided at the VOL plugin init phase, all objects should be created by the same process, or they should always be created collectively using `H5Pset_all_coll_metadata_ops()`.

B. RADOS VOL Plugin

The RADOS VOL plugin currently supports a subset of HDF5 features including groups and datasets. I/O support includes partial I/O and datatype conversion. The RADOS plugin was developed following many of the lessons learned while developing the DAOS plugin and uses that plugin's code as the base to work from.

1) Interface: The interface for the RADOS plugin is simpler than that of the DAOS plugin since it does not need to internally handle different version epochs. The RADOS plugin does, however, retain the same behavior as the DAOS plugin regarding independent vs collective I/O. Therefore, raw data operations are always independent, and all metadata operations are independent unless collective is requested.

2) Mapping and Object IDs: In the RADOS plugin, HDF5 objects are again stored as RADOS objects; though, unlike in the DAOS plugin, objects for multiple files are not separated by an abstraction in the object store, as there is no analog to DAOS containers in RADOS. All HDF5 objects are stored in a single pool. However, since RADOS object IDs are null-terminated strings, we have more flexibility in creating object IDs. To avoid conflicts between different files, all object IDs are prefixed with the HDF5 file name for the file they belong to, then a numerical ID within the file (described below) is appended. In the future we plan to implement an *object index* object for each file to list all the objects in a file, to be used when deleting a file, a *file index* object for each pool listing all the files in the pool, and a *global metadata* object similar to that used by the DAOS plugin.

RADOS objects contain both a linear byte array and a key-value store. Constant metadata for HDF5 objects is serialized, placed together, and stored directly in that objects byte array. For HDF5 groups, links are stored in the key-value store, where the key is the link name and the value is a 64-bit binary form of the numerical object ID. This binary ID can be converted to the string form used to identify the object by rendering it in hexadecimal and prepending the file name. HDf5 dataset chunks are stored in separate RADOS objects, where the object ID is the same as that of the dataset containing the chunk, postfixed with the chunk coordinates. In the future, we will need to change this scheme slightly to eliminate the (unlikely) possibility of conflicts between files involving chunks. HDF5 attributes are not yet implemented, but they are planned to be added to the objects key-value store, with the attributes value stored directly in the key-value stores value.

The RADOS plugin currently has the same restrictions involving object ID allocation as the DAOS plugin. All HDF5 objects must therefore be created collectively or by a single process. In addition, since the global metadata object is not implemented yet, all objects must be created during the course of a single file open. We plan to address both of these limitations in the future. We believe we can implement independent object ID allocation using the existing features of RADOS.

C. SWIFT VOL Plugin

The Swift VOL plugin currently supports basic HDF5 functions, including file and group creation, dataset create/open/close, and dataset read/write.

1) Mapping: One of our design considerations was to preserve the HDF5 file's hierarchy in Swift's flat namespace, such that path traversal and future data re-formatting would be easier. We leveraged the uniqueness of the HDF5 object name within a HDF5 file. For example, a dataset object in HDF5 file has a absolute path name, e.g., '/group/subgroup/dataset'. All HDF5 objects are named with its absolute path name in the Swift container, so that the original HDF5 hierarchy is easily reversible based on the object name, at the expense of considerable cost if an intermediate group is renamed or deleted.

An HDF5 group is mapped as a sub-container in Swift, and all HDF5 objects within the same group are kept in the same sub-container. This reduces the query cost during the IO request. Since Swift does not support nested containers, an empty object is created in the parent container with the same name of the sub-container, such that the sub-group is linked with the parent group without needing to store additional index or metadata table for tracking the objects within a file. All HDF5 attributes and user defined metadata are appended in the object's extended attributes.

2) Implementation: The Swift VOL plugin is written in C and Python. We chose Python because Swift is better supported in Python than other languages, e.g., the last update to the C interface for Swift is more than five years ago. The HDF5 C functions are translated by the Swift VOL into Swift's Python interface, e.g., swift.upload. Implementing the VOL on a Python layer immediately exposed us some performance reductions (e.g., memory copy overhead between C layer and Python layer), but on the other hand, largely simplified our implementation cost. We will discuss more details regarding the object store I/O internals in Section VI-F.

VI. Evaluation with HPC Applications

Three HPC applications were selected to evaluate the object stores. The applications are: VPIC, H5Boss, and BDCATS and covered I/O patterns that include both large sequential I/O and random small I/O, which are representative in traditional and modern HPC workloads.

A. Sequential and Large Write Benchmark Tests

In this section, we evaluate the VPIC benchmark, which performs large sequential data writes. The VPIC-IO kernel is extracted from a plasma physics code called VPIC [20], which simulates kinetic plasmas in multi-dimensional space. In this kernel, each MPI process writes a fixed number of particles. VPIC data structures use eight 1-D arrays to represent the eight variables associated with each particle. By fixing the number of particles written by each process, we conducted weak scaling tests. This benchmark also represents a majority of HPC workloads, e.g., climate and physics, that have large and contiguous I/O pattern.

The first test is single node test with 1 to 32 processes. We measured the I/O bandwidth on the four storage systems. As shown in Figure 3, DAOS largely outperforms all other storage systems by writing to tmpfs-emulated NVM. RADOS and Swift show relatively low bandwidth due to hardware limits (testbed network bandwidth is only 1GB/sec), as described in Section III. RADOS is better than Swift due to the fact that we set the replication size as 1 in RADOS, but are not able to change the Swift configuration. RADOS also uses the librados API to communicate directly with the server (OSD) from the client, while Swift can not bypass the gateway nodes with the existing API. All three object

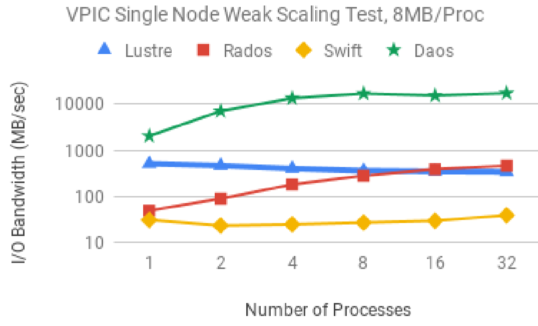

Fig. 3. VPIC Single Node Weak Scaling

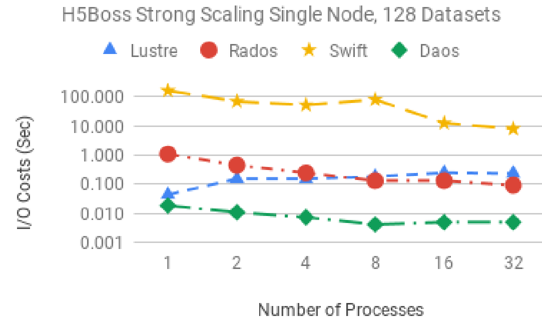

Fig. 5. H5Boss Single Node Strong Scaling

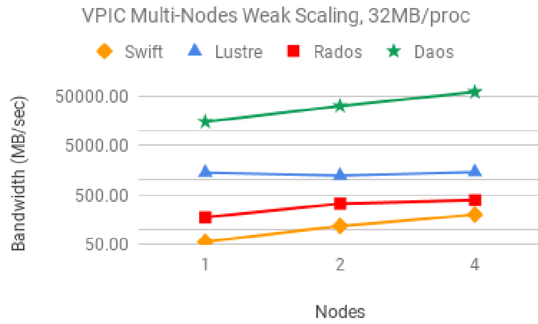

Fig. 4. VPIC Multi-Nodes Weak Scaling

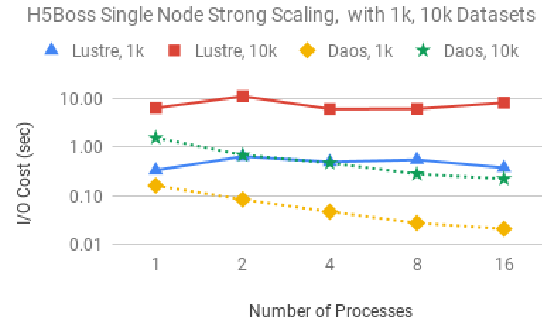

Fig. 6. H5Boss Single Node Strong Scaling

stores, however, reveal better scalability than the traditional POSIX-compliant Lustre filesystem.

Similar trends are seen in the multi-node test with 1 to 4 nodes (workload per process is set as 4 times larger than single node test, which is now 32 MB per process, 8 processes per node), in Figure 4. The scalability of RADOS and Swift are slightly better than Lustre, with DAOS as winner in all tests.

B. Random and Small Write Benchmark

Random and small I/O is not rare in HPC, for example, in astronomy field, the collected data are stored in millions of small FITS files. Scientists submit a list of (plate, mjd, and fiber) IDs to locate a targeted astronomy object. In case of BOSS (Baryon Oscillation Spectroscopic Survey), the Sloan Digital Sky Survey (SDSS) project 2, the total number of files is 276,575, with each file containing 1000 datasets. The data provides information about the composition of stars and galaxies, and can be used to obtain their redshift, e.g., how fast a star is moving away from the earth.

A query submitted with H5Boss [35] can transfer multiple files and randomly pick certain amounts of datasets from each file. During output, the I/O transaction size becomes relatively small, ranging from several hundreds KBs to a few MBs. We tested the H5Boss benchmark with various queries, ranging from 128 datasets to 10240 datasets query.

We evaluated the three object stores with H5Boss strong scaling tests on a single node. The first test writes 128 datasets separately on those storage testbeds. As shown in Figure 5, the I/O cost on object stores keeps decreasing as we increasing the number of processes. Lustre (the blue line), however, does not decrease and suffers from POSIX constraints and file system locking overhead as we scale.

With current VOL design, each HDF5 dataset is chunked and each chunk is written to an independent object in the object store. For example, in case of 32 processes, each dataset will be evenly split into 32 objects. With 128 datasets, the number of objects is 4096. When we increase the workload from 128 datasets to 1024 and 10240 datasets, shown in Figure 6 for Lustre and DAOS, Swift and RADOS failed with various errors. We have not spent enough time to investigate the issue, but believe the network concurrency and bandwidth limit on our testbed are possible causes. DAOS continue to scale with this high number of concurrent datasets/objects, and Lustre can handle these large amounts of dataset creation/writing, but does not scale well and the latency is higher.

It is obvious that such random and small I/O pattern is not friendly to all tested storage systems. With 128 and 10240 datasets written into one file separately, the average performance ratio between Lustre and DAOS is reduced from 30X to 18X.

On multi-node strong scaling tests, Figure 7, DAOS is able to handle 60k datasets per second with 4 client nodes, while Lustre only manages to process 7.5k datasets per second with

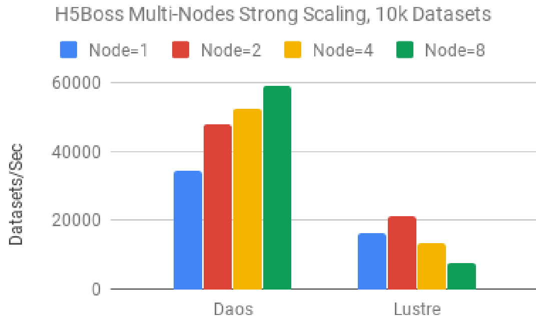

Fig. 7. H5Boss Multi-Node Strong Scaling, Lustre v.s. DAOS

same number of clients and servers.

C. Large and Sequential Read Benchmark

The last benchmark is BD-CATS. This I/O kernel is extracted from a parallel clustering algorithm, used for analyzing data produced by particle simulations, such as VPIC. In this kernel, data related to the particles are read by all the MPI processes with a load-balanced distribution. Each process reads a contiguous chunk of data from different sub-regions.

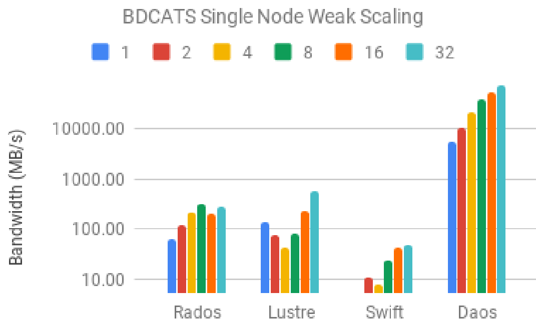

Fig. 8. BDCATS Single Node Weak Scaling

In Figure 8, scalability of Lustre is better than its corresponding write test in Figure 3, probably due to less file system locking in this read pattern, and the benefit of Lustre readahead. RADOS is faster than Swift not only in the previous write test, but also in this read test, i.e., 12X faster on average. This is partly because RADOS supports partial reads on object and the librados library can directly connect from the client to the server, while in Swift, reading a portion of an object is not supported by its interface, and the gateway node can not be bypassed.

D. Object Store Performance Optimization

Object stores, like parallel file systems, including Lustre, have options for users to tune the I/O performance. In this section, we discuss some initial explorations in object store I/O tuning.

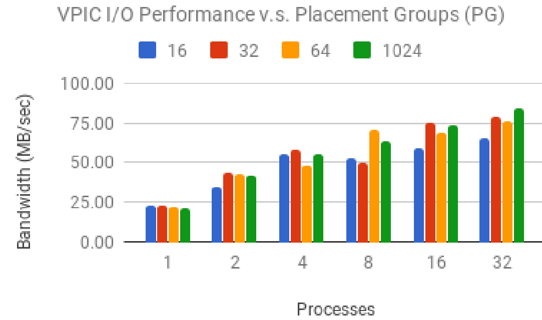

Fig. 9. Performance Variance with Different Placement Groups

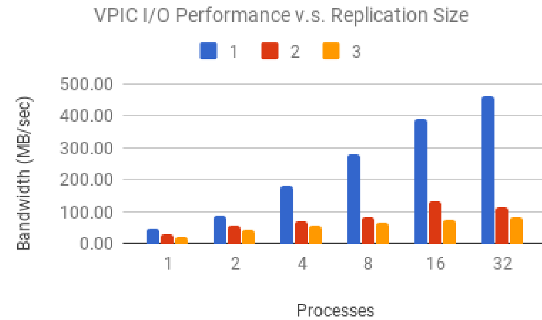

Fig. 10. Performance Evaluation with Different Replication Size

The first test, Figure 9, tunes the number of placement groups on RADOS. A placement group (PG) aggregates objects within a pool, because tracking object placement and object metadata on a per-object basis is computationally expensive. We varied the placement group size from 16 to 1024 and the number of processes from 1 to 32. Among all tests, the initial setting of 16 placement groups was the worst case. For example, in the case of one process, maximum performance is achieved with 32 placement groups, and on 32 processes, the maximum is achieved with 1024 placement groups. With more than 16 placement groups, current tests do not show any other clear pattern regarding optimal numbers of placement groups.

We also varied the replication size in RADOS, as shown in Figure 10. The replication size determines how many copies the RADOS will make for each data object. The default value is 3, which is a common strategy in object stores. RADOS provides users with an option to either choose erasure coding or replication when creating a new pool. For most HPC applications, data does not need to be replicated, especially for simulation data, as it can be reproduced instead. From this test, we can see that trading durability for performance is possible by reducing the number of replications.

E. Impact on HPC Application Source Code

The HDF5 VOL plugins developed here aimed to provide HPC applications with a transparent transition to object

stores. We summarize the number of lines changed to the three applications in the following table:

	VPIC	H5Boss	BDCATS
Swift	7	6	7
RADOS	7	7	7
DAOS	4	4	4

Apart from these very modest amounts of code modifications, RADOS and Swift also require users to create an account on the storage system, for authentication purposes.

F. Object Store I/O Internals

Through the development of VOL plugins and the evaluation of the object stores, we found several interesting facts about the object stores' internal operation. These findings may lead us to better customize or design object stores for HPC applications in the future. The findings include but are not limited to the following points:

1) Most object stores are designed to only handle I/O on entire objects, instead of finer granularity I/O, such as provided by POSIX, which is required by HPC applications.
2) Swift does not support partial I/O on object. Although it supports segmented I/O on large objects, the current API can only read/write an entire object. This stops us from performming parallel I/O with chunking support in HDF5.
3) RADOS offers librados for clients to directly access its OSD (object storage daemon), which is a performance benefit as the gateway node can be bypassed.
4) Mapping HDF5's hierarchical file structure to flat namespace in object store will require additional utility tools for users to easily view the file's structure.
5) Traditional HPC I/O optimization techniques may be applied in object stores, for example, two-phase collective I/O, as currently each rank issues the I/O to object independently. A two-phase collective I/O-like algorithm is possible when considering the object locality.
6) Object stores trade performance for durability. Reducing the replication size (default is frequently 3) when durability is not a concern for HPC application can increase the bandwidth.

VII. Related Work

There are a few related works evaluating object store for HPC applications. Among them, MarFS [10] proposed a scalable near-POSIX file system by using one or more POSIX file systems as a scalable metadata component and one or more data stores (object, file, etc) as a scalable data component. Early evaluation of MarFS [27] has shown excellent scalability and parallel I/O performance, but it is not clear how traditional HPC applications will perform on MarFS. Our work is the first work that comprehensively evaluated object store technologies with actual representative HPC I/O workloads.

CERN is one of the earliest adopters of object stores in the HPC community. The proposed Davix project [5] aims to make file management over HTTP-based protocols simple. Davix focuses on high-performance remote I/O and data management of large collections of files, with support for WebDAV, Amazon S3, Microsoft Azure and HTTP access methods. The project developed libdavix, a C++ library that offers an HTTP API, a remote I/O API and a POSIX compatibility layer. The difference between Davix and our evaluation is that our work translates application operations (through HDF5 API calls) directly into object store operations, rather than translating POSIX I/O to non-POSIX I/O, and the underlying storage leverages those object-oriented solutions, e.g., scalable hashing mechanisms. Davix's focus is HTTP-based file management, while our main interest is evaluation of HPC I/O performance on object stores.

Openstack Swift's I/O is based on HTTP, optimization techniques, including threading and RDMA support, have shown to largely improve I/O bandwidth. For example, in [39], the author showed how modifying the Swift threading model can achieve 18% mean improvement in performance with objects larger than 512 KiB and obtain a similar performance with smaller objects. Swift-X [31], on the other hand, accelerated several synthetic applications up to 7.3x with RDMA. This work aligns with our finding that a HPC-oriented object store will need both software optimization as well as HPC friendly hardware acceleration.

As mentioned earlier, HDF5's object-based data model is a good fit for object storage. Researchers in [40] explored scientific application performance with HDF5-emulated object storage. The work is interesting in terms of connecting HDF5 applications directly with HDF5-emulated storage. However, the actual object store impact to HPC application is not known from this work. Our evaluation is based on existing object stores and the selected applications represent a majority of actual HPC I/O patterns.

VIII. Conclusion and Future Work

Parallel file systems' high performance is a result of decades of research and development effort. Object store technologies, however, are still missing capabilities required for HPC environments. In this work, we developed three HDF5 Virtual Object Layer plugins for three different object stores, Ceph/RADOS, Openstack Swift, and Intel DAOS. The selected applications/benchmarks covered a majority of typical HPC I/O patterns. By strategically choosing the right I/O layer to adapt to the object store, only an average of six lines of code changes to existing application code was required to target the object store, and we were able to run the HPC application on an object store simply by re-compiling after this minor change.

Through the evaluation, we demonstrated that object stores have better scalability than POSIX file systems like Lustre, in both large contiguous write and random small write. We discovered important API differences between RADOS and Swift, e.g., RADOS has better support for partial read/write, which enables us to implement a better parallel I/O solution.

Among all object stores we evaluated, Intel DAOS has demonstrated outstanding performance by leveraging next generation NVMe/SCM technologies, although our results should be followed up when the tmpfs-emulated SCM can be replaced with actual SCM.

Given more time, we would like to go further to optimize the object I/O, but we believe the existing HPC I/O optimization techniques could be applied to object store, e.g., two-phase collective I/O. The lessons learned in this work are beneficial to future object store development for HPC, and for HPC users to have a better understanding of and preparation for object stores.

IX. Acknowledgement

We appreciate the resources provided by National Energy Research Scientific Computing Center and Intel Corporation. We also thank HDF Group for extensive support in the VOL design and implementation.

References

[1] ALS, https://als.lbl.gov/.

[2] Cori at NERSC, http://www.nersc.gov/users/computational-systems/cori/.

[3] Cori File System, http://www.nersc.gov/users/storage-and-file-systems/file-systems/ngfdrawings/.

[4] Daos Community, https://wiki.hpdd.intel.com/display/dc/daos +community+home.

[5] DAVIX, http://dmc.web.cern.ch/projects/davix/home.

[6] HDF5 Benchmarks at NERSC, https://github.com/valiantljk/h5spark/tree/master/mpiio.

[7] IBM Spectrum Scale Version 4 Release 2.3 Administration Guide, https://goo.gl/wa2wew.

[8] Intel Mpi Benchmarks, https://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[9] LCLS-II, https://lcls.slac.stanford.edu/lcls-ii.

[10] MarFS, https://github.com/mar-file-system/marfs.

[11] OpenStack Swift, https://docs.openstack.org/swift/latest/.

[12] POSIX Extensions, http://www.pdl.cmu.edu/posix/.

[13] Posix must die, http://www.linux-mag.com/id/7711/comment-page-1/.

[14] RADOS, https://ceph.com/geen-categorie/the-rados-distributed-object-store/.

[15] Theta at ANL, https://www.alcf.anl.gov/theta.

[16] VPIC, https://github.com/lanl/vpic.

[17] What's So Bad About POSIX I/O, https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/.

[18] T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. He, T. Kurth, T. Koskela, M. Lobet, T. Malas, L. Oliker, A. Ovsyannikov, A. Sarje, J. L. Vay, H. Vincenti, S. Williams, P. Carrier, N. Wichmann, M. Wagner, P. Kent, C. Kerr, and J. Dennis. Evaluating and optimizing the nersc workload on knights landing. In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 43–53, Nov 2016.

[19] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook ' s photo storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, number October, pages 1–8, Vancouver, BC, 2010.

[20] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh Performance Three-dimensional Electromagnetic Relativistic Kinetic Plasma Simulation. *Physics of Plasmas*, 15(5), 2008.

[21] Peter J. Braam. Scalable locking and recovery for network file systems. In *Proceedings of the 2nd International Workshop on Petascale Data Storage - PDSW'07*, page 17, New York, New York, USA, 2007. ACM Press.

[22] M. Scot Breitenfeld, Neil Fortner, Jordan Henderson, Jérome Soumagne, Mohamad Chaarawi, Johann Lombardi, and Quincey Koziol. DAOS for extreme-scale systems in scientific applications. *CoRR*, abs/1712.00423, 2017.

[23] M Scot Breitenfeld, Quincey Koziol, Neil Fortner, Jerome Soumagne, and Mohamad Chaarawi. Use of a new I/O stack for extreme-scale systems in scientific applications. In *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, 2016.

[24] S. Byna, J. Chou, O. Rubel, Prabhat, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K. Hsu, K. Lin, A. Shoshani, A. Uselton, and K. Wu. Parallel i/o, analysis, and visualization of a trillion particle simulation. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2012.

[25] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11. IEEE, may 2009.

[26] Raghunath Raja Chandrasekar, Lance Evans, and Robert Wespetal. An Exploration into Object Storage for Exascale Supercomputers. In *Proceedings of the 2017 Cray User Group*, 2017.

[27] H. Chen, G. Grider, and D. R. 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, April 2017.

[28] 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, apr 2017.

[29] Nikita Danilov, Nathan Rutman, Sai Narasimhamurthy, and John Bent. Mero: Co-Designing an Object Store for Extreme Scale. In *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, 2016.

[30] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGPLAN Not.*, 33(11):92–103, October 1998.

[31] S. Gugnani, X. Lu, and D. K. Panda. Swift-x: Accelerating openstack swift with rdma for building an efficient hpc cloud. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 238–247, May 2017.

[32] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J. Wright. Architecture and Design of Cray DataWarp. In *Proceedings of the 2016 Cray User Group*, London, 2016.

[33] Cengiz Karakoyunlu, Dries Kimpe, Philip Carns, Kevin Harms, Robert Ross, and Lee Ward. Toward a unified object storage foundation for scalable storage systems. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8. IEEE, sep 2013.

[34] Wei-keng Liao and Alok Choudhary. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, number November, pages 1–12. IEEE, nov 2008.

[35] J. Liu, D. Bard, Q. Koziol, S. Bailey, and Prabhat. Searching for millions of objects in the boss spectroscopic survey data with h5boss. In *2017 New York Scientific Data Summit (NYSDS)*, pages 1–9, Aug 2017.

[36] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. DAOS and Friends: A Proposal for an Exascale Storage System. In John West, editor, *2016 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 585–596, Salt Lake City, 2016.

[37] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, Aug 2003.

[38] Shadi A Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H Campbell. Ambry: LinkedIn's Scalable Geo-Distributed Object Store. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*, pages 253–265, New York, New York, USA, 2016. ACM Press.

[39] R. Nou, A. Miranda, M. Siquier, and T. Cortes. Improving openstack swift interaction with the i/o stack to enable software defined storage. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, pages 63–70, Nov 2017.

[40] R. Karim E. Laure S. W. Chien, S. Markidis and S. Narasimhamurthy. Exploring scientific application performance using large scale object storage. In *arxiv 2018*, 2018.

[41] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, California, USA, 2002.

[42] Joost Snijder, Jan M. Schuller, Anika Wiegard, Philip Lössl, Nicolas Schmelling, Ilka M. Axmann, Jürgen M. Plitzko, Friedrich Förster, and Albert J. R. Heck. Structures of the cyanobacterial circadian oscillator frozen in a fully assembled state. *Science*, 355(6330):1181–1184, 2017.

[43] Stephen Sugiyama and David Wallace. Cray DVS: Data Virtualization Service . In *Cray User Group Meeting (CUG2016)*, 2016.

[44] M. Vilayannur, Samuel Lang, Robert B. Ross, R. Klundt, and L. Ward. Extending the posix i/o interface: A parallel file system perspective. (ANL/MCS-TM-302), 2008.

[45] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[46] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2006.