Lessons Learned from the Fast Forward Storage and I/O Project

Blinded for review
,

***Abstract*—The DOE Extreme-Scale Technology Acceleration Fast Forward Storage and IO Stack project is going to have significant impact on storage systems design within and beyond the HPC community. With phase two of the project starting, it is an excellent opportunity to explore the complete design and how it will address the needs of extreme scale platforms. This paper examines each layer of the proposed stack in some detail along with cross-cutting topics, such as transactions and metadata management.**

This paper not only provides a timely summary of important aspects of the design specifications but also captures the underlying reasoning that is not available elsewhere. We encourage the broader community to understand the design, intent, and future directions to foster discussion guiding phase two and the ultimate production storage stack based on this work. An initial performance evaluation of the early prototype implementation is also provided to validate the presented design.

I. Introduction

Current production HPC IO stack design is unlikely to offer sufficient features and performance to adequately serve extreme scale science platform requirements. While new hardware, such as non-volatile memory will help, we still need a new software stack to incorporate this new hardware as well as address the extreme parallelism and performance requirements demanded by exascale applications. Adding to the problem complexity is the variety of Big Data problems users want to address using these platforms. Unlike the centralized storage arrays favored for HPC platforms, big data analytics systems have grown up using storage distributed on all of the nodes driving a very different software architecture. With post-exascale platforms required to address both workloads, a new storage stack is required.

To address these challenges, a joint effort between the US Department of Energy's Office of Advanced Simulation and Computing and Advanced Scientific Computing Research commissioned a project to develop a design and prototype for an IO stack suitable for the extreme scale environment. It will be referred to as the Fast Forward Storage and IO (FFSIO) project. This is a joint effort led by Lawrence Livermore National Laboratory, with the DOE Data Management Nexus leads Rob Ross and Gary Grider as coordinators and contract lead Mark Gary. The participating labs are LLNL, SNL, LANL, ORNL, PNL, LBNL, and ANL. Additional industrial partners contracted include the Intel Lustre team, EMC, DDN, and the HDF Group. This team has developed a specification set [10] for a future IO stack to address the identified challenges. The first phase completed in 2014 with a second phase currently getting underway. The core focus of the first phase was basic functionality and design. While an idealized potential system would be the perfect target architecture, the reality of budgets has tempered many of the decisions. For example, extensive availability of NVRAM or SSDs on all of the compute nodes is currently not economically feasible limiting some of the potential design choices. With this in mind, the second phase will refine this design incorporating fault recovery and other features missing from the first phase.

The complete design seeks to offer high availability, byte-granular, multi-version concurrency control. Through the use of a copy-on-write style mechanism reducing storage space and writing time requirements, multiple versions of an object can be stored efficiently. By assuming the client interface will be through an IO library, a more complicated interface offering richer functionality can be incorporated while requiring only minimal end-user code changes. Managing most data access in a platform-local layer rather than requiring writing to centralized storage will better support the performance and energy requirements of extreme scale application compositions.

Overall, the architecture shifts from the idea of files and directories to containers of objects. This shift avoids the bottlenecks related to the POSIX files and directories structure such as the serialization of file creation, the limitation and impact of the number of files in a directory, and the limited semantics of a byte stream. Instead, the new interface focuses on high-level data models and their properties and relationships. This concept permeates the entire IO stack.

In addition to addressing the traditional scientific workload, this project seeks to expand functionality to better support Big Data type applications. The key idea is to support Arbitrary Connected Graphs (ACGs) such as those used in Map-Reduce systems. Key system features are introduced to efficiently support these computing models in addition to the typically bursty IO loads of more traditional HPC applications.

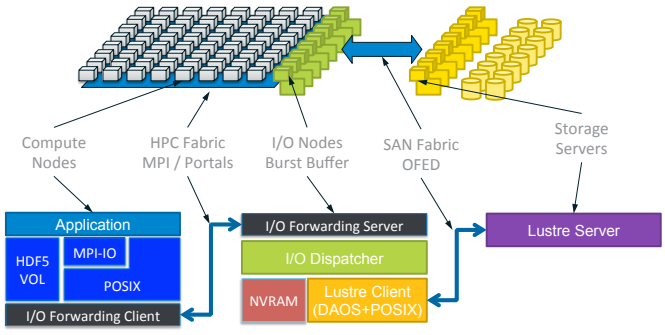


Fig. 1. Target Architecture and Component Mapping

At a more detailed view, the various layers of the IO stack each contribute different functionality. The architecture (Figure 1) incorporates five layers, some of which have potentially optional components. The top layer is comprised of

generally a high level IO library, such as the demonstration HDF5 library [30] and a more complex API for accessing the lower level components. This layer is in dark blue. Becuase the system supports more complex architectures and supports richer functionality, hiding this complexity behind a user-friendly API is the intent. It is possible to access the storage stack through the more complex API, but the additional requirements beyond standard POSIX calls will prompt most users to use an IO library. This layer incorporates the necessary features for ACGs from a end-user's perspective.

Below the user API is an IO forwarding layer that redirects IO calls from the compute nodes to the IO dispatching layer (in black). This IO forwarding layer is analogous to the function of the IO nodes in a BlueGene machine or the passive data staging processes demonstrated previously [21], [2]. One special function of note for this layer is that it is where function shipping will be deployed. This is discussed in Section III. The next two layers have considerable functionality.

The IO Dispatcher (IOD) serves as the primary storage interface for the IO stack (in green) and offers features like Burst Buffers to insulate the persistent storage array from bursty IO workloads. Ideally, the IOD layer's functionality can be optional based on available hardware and compute power provided on the IO Nodes (IONs). Transactions are handled primarily at this layer. Much of the functionality offered at this layer would shift either up or down the stack as discussed in detail below.

The Distributed Asynchronous Object Storage (DAOS) layer serves as the persistent storage interface and translation layer between the user-visible object model and the requirements of the underlying storage infrastructure. Transactions work a bit differently at this layer and are called epochs to distinguish them. DAOS is intended to be the traditional file system-like foundation on which everything else is built with no dependence on any technologies specified above it (in dark pink and yellow). For example, the IOD layer with or without burst buffers is not required for DAOS to operate properly. Instead, the DAOS layer can handle all of the IO operations from the user API layer, albeit with the potential performance and usability penalty of manipulating the shared, persistent storage array directly.

At the bottom is the Versioning Object Storage Device (VOSD) (in purple). It serves as the interface for storing objects of all types efficiently for each storage device in the parallel storage array. Think of this layer as the physical disk interface layer. In terms of Lustre, this would replace the API on individual storage devices with an interface friendlier to the containers of objects and transactions/epochs concepts used in the higher layers.

Along with analysis of the published design documents, a discussion of the design philosophy representing the overall intent is presented. This information represents information that may or may not have been written down, but is the intent of ultimate system. This information was gathered through interviews with the core FFSIO team members. These ideas are presented to give a fuller picture of where the project is going rather than dwelling on any limitations of the published documents. This is most important to illustrate how different concepts will work across layers since that information is spread across multiple documents and may lack a cohesive overall view.

The rest of the paper is organized as follows. An overview of related work is presented first in Section II. Section III discusses the programmatic interface end users will see when interacting with the storage array. This will be discussed in the context of the HDF5 based example library used for the functionality demonstration. Section IV briefly discusses the motivation and proposal for the IO forwarding layer. Section V describes the IO Dispatcher layer and the broad functionality it offers. This will detail the pieces of the layer that are potentially optional and mention the cross-cutting features discussed in a later, cross-cutting section. Section VI discusses how the DAOS layer functions. As with the IOD layer, the cross-cutting features will be mentioned, but discussed more fully in the cross-cutting section. The VOSD layer is discussed in Section VII. In particular, the mapping between the DAOS and VOSD layers are explored as it pertains to the physical storage. Next is an exploration of cross-cutting features like transactions and metadata management in Section VIII. Since these and other features are spread across multiple layers, it makes more sense to discuss them independently once an understanding of the overall structure has been presented. A demonstration of the functionality is presented in Section IX. This shows that the prototype system based on the proposed design can function. The paper is concluded in Section X with a summary of the broad issues covered in the paper.

II. Related Work

Many projects over the last couple of decades have sought to address some challenging aspect of parallel file system design. The recent rise of "Big Data" applications with different characteristic IO patterns have somewhat complicated the picture. Vendors are shifting products to address the far larger market forcing HPC systems to adapt to these different storage approaches. Extreme scale machines will be expected to handle both the traditional simulation-related workloads as well as applications more squarely in the Big Data arena. This will require some adjustments to the underlying system for good performance for both scenarios.

The major previous work is really limited to full file systems rather than the mountain of file system refinements made over the years. A selection of these other file systems and some features that make it relatively unique are described below.

Lustre [7] is the de facto standard on most major clusters offering scalable performance and fine-grained end-user and programmatic control over how data is placed in the storage system. The broad community support has led to a solid code base with sufficient optimizations to serve as the low-cost, proven solution. For each installation, system-wide settings that apply to all files on the file system are made. The end user, should they have different needs can reconfigure these characteristics on a file-by-file basis. This becomes an issue because the dominant file size is tiny. In many cases, it can be < 4 KB. To keep from slowing the overall system performance when creating and opening these files, most systems are configured to use a 1 MB stripe size and a stripe count of 4 meaning only 4 storage targets are used per file. This limits the default aggregate bandwidth to the combined speed of four storage targets. By reconfiguring on a file-by-file basis, this default can be overcome for large, parallel files achieving very high performance. The downside is that this setting must

be done to take advantage of the full parallel file system performance.

Ceph [32] is a distributed object store and file system. It offers both a POSIX and object interface including features typically found in parallel file systems. Ceph's unique striping approach uses pseudo-random numbers with a known seed eliminating the need for the metadata service to track where each stripe in a parallel file is placed. Ceph's strengths are in providing good perforamnce and scalability with the ability to handle failures and deploying new storage adapting the system in a live environment. However, this failure handling advantage was shown [31] to limit peak performance more than other systems like Lustre. More recently, some of these limitations have begun to be addressed by the Ceph team, but no new evaluations have been performed to determine if these changes close the gap sufficiently to address the extreme HPC performance needs.

PVFS [8] was built understanding the scalibility bottlenecks Lustre suffers. For example, Lustre requires all processes opening a file to hit the metadata server to receive a proper file handle. PVFS reduces this load by allowing a single process to open a file and sharing the handle with other processes participating in the IO operation. There are also other optimizations that enhance file system performance. It has been commercialized in recent years as OrangeFS.

GPFS [27] offers a highly scalable parallel file system with robust functionality to handle both parallel storage, recovery, and optimization. It only supports a hands-off approach for providing good performance for scaling parallel IO tasks and is used extensively by its owner, IBM. Unfortunately, the stripe size is fixed introducing potentially false sharing, when two processes indpendently write to the same stripe, but without overlapping, causing potentially reduced performance. Beyond these sorts of fixed parameters, a wide variety of optimizations and features are available for additional licensing fees.

Panasas [24] uses a fundamentally different approach to parallel file system performance. When parallel writers simultaneously write to a shared file, the system dynamically adapts the number of stripes to maintain high performance. This adaption is invisible to the user other than seeing that the system maintains high performance no matter what configuration the workload exhibits.

This project learns from all of these parallel file system efforts and offers a scalable approach that can work well for everything from a small cluster to the largest exascale platforms. By understanding what works well and what the limitations are for each of the above systems as well as emerging hardware architectures, this project addresses the limitations while maintaining the advantages of the above systems.

Other file systems, like GoogleFS [11] and HDFS [28], address distributed rather than parallel computing and cannot be compared directly. The primary difference between distributed and parallel file systems is the ability of the file system to store and retrieve data simultaneously from multiple clients, in parallel, and treat the resulting collection of pieces as a single object. Distributed file systems rely on a single client creating a file typically on a single storage device. For performance, the file or object may be replicated. The other, popular distributed file system of note is NFS [25] that has been used for decades for enterprise file systems. NFS is known to support a global namespace with data migrating towards users on access and pushed towards safer storage based on local platform characteristics. These other file systems are mainly of interest in the context of the ACG features of FFSIO and will be discussed more in Section VIII-D.

The main alternative from scratch design for a file system is Sirocco [9]. Rather than continuing the striped design of existing parallel file systems, Sirocco is inspired by peer-to-peer and object-based systems and includes features like transactions to protect data modification process independence when writing to avoid coordination overhead. The base assumptions are that storage is pervasive and volatile. Storage devices and locations may come and go randomly, reminiscent of the Google or Ceph assumptions of regularly failing hardware. When data is pushed into the system, initial resilience characteristics are guaranteed prior to returning control back to the user. Then, as system pressures dictate, data will either replicate as demanded by use and/or migrated towards long-term resilience requirements. Unlike the FFSIO project, Sirocco assumes it is possible that data may be successfully stored in the system, but it is currently inaccessible because all copies are currently offline. There is also some potential difficulty in finding data since it will migrate around the system. To be fair, Sirocco intends to function as the storage layer for a higher level file system API making many of the awkward system features invisible to the end user. Sirocco is in the process of being released publicly.

III. End-User API Layer

Since the proposal specifies a high-level IO API will be the primary end-user interface for programmatically interacting with the FFSIO stack, the team used the HDF5 API and leveraged its Virtual Object Layer (VOL) for the initial design and implementation demonstration. This also serves as a good test determining what are strictly necessary extensions to an existing IO API to support the new functionality. The additional functionality, such as transactions, can be ignored for legacy implementations, but these applications will not be able to take advantage of the asynchronous IO support inherent to the new API. The additions comprise (Figure 2):

1. API extensions to support new functionality provided by the FFSIO project. This includes calls for managing asynchronous request lists, performing asynchronous operations, creating and managing transactions, end-to-end data integrity, and data type and functions to support the ACG functionality more efficiently than the current API.
2. Function shipping from Compute Nodes (CN) to IO Nodes (ION). This provides the application developer with the capability of sending computation down to the IONs and get back results and perform other operations such as indexing and data reorganization for more efficient retrieval.
3. Analysis Shipping from compute nodes to IO Nodes or DAOS nodes. This is similar to function shipping, but instead of returning the result over the network, it is stored on the nodes and pointers to the data are returned.

Function and Analysis Shipping are part of the cross-cutting features and are discussed in Section VIII-C.

HDF5 [30] has a versatile data model offering complex data objects and metadata. Its information set is a collection of datasets, groups, datatypes and metadata objects. The data

model defines mechanisms for creating associations between various information items. The main conceptual components for data stored in HDF5 are described below.

- **File**: In the HDF5 data model, the collection of data items stored together is represented by a file. It is an object collection that also describes the relationship between them. Every file begins with a root group "/" serving as the "starting-point" in the object hierarchy.
- **Group**: A group is an object allowing association between HDF5 objects. It is synonymous with directories in a file system. A group could contain multiple other groups, datasets, datatypes or attributes within it. Groups are named and then accessed using a standard path notation similar to Linux with a "/" separating each group name in the hierarchy from the root to the nested group of interest.
- **Dataset**: HDF5 datasets are objects representing actual data or content. Datasets are typically arrays with potentially multiple dimensions. Other types, such as strings and scalars, are also possible. A dataset is characterized by a dataspace and a datatype. The dataspace captures the rank (number of dimensions) and the current and maximum extent in each dimension. The datatype describes the type of its data elements. By default, the entire data set is stored as a single chunk reassembled from all processes. It is possible to use a uniform chunking format where data is stored in fixed sized chunks instead.
- **Attribute**: Attributes are used for annotating HDF5 objects. They are datasets themselves and are attached to existing objects.

A. Virtual Object Layer

The Virtual Object Layer is an abstraction mechanism internal to the HDF5 library [30]. As shown in Figure 2 it is implemented just below the public API. The VOL exports an interface that allows writing plugins for HDF5 enabling developers to handle data in ways other than writing to storage in an HDF5 format. Plugin writers provide an implementation for a set of functions and are trusted to provide the proper semantics for the new environment. For example, data staging could be implemented in the VOL layer by replacing writing to disk in the HDF5 format to sending data to a data staging area using some messaging mechanism.

For this project, rather than the default writing to disk in the HDF5 format, the VOL is used to interact with the IOD layer and the different concepts it offers without requiring all of the functionality be exposed to users. For example, the containers and objects concept is transparently mapped to the files and datasets existing HDF5 users are familiar with. This reduces the difficulty porting applications to the new IO stack.

The IOD VOL plugin serves as the bridge between HDF5 and the IOD Layer (Figure 2). The application calls the HDF5 library while running on the system's compute nodes. Using the VOL architecture, the IOD VOL plugin uses a function shipper (RPC library) to forward the VOL calls to a server component running on the IO nodes (IONs). This function shipping is the IO Forwarding Layer discussed briefly in Section IV. Once the calls arrive at the IO nodes, they are translated into IO Dispatcher (IOD) API calls and executed at the IONs.

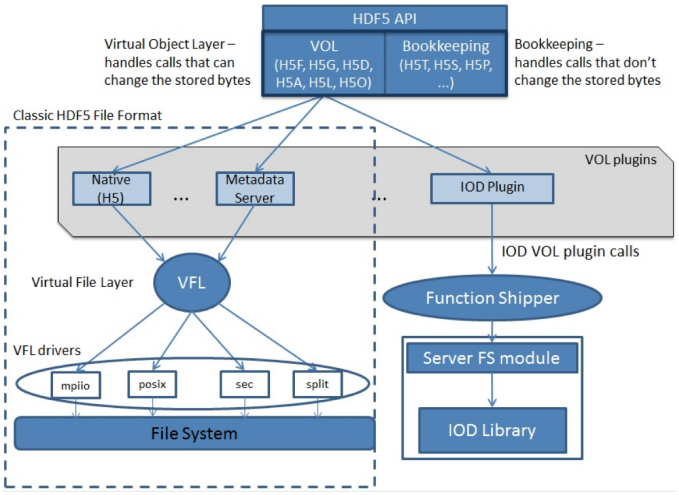


Fig. 2. Architectural view of the VOL abstraction mechanism

Since the IOD layer is optional by design, a second VOL plugin is required to access DAOS directly. This additional complexity was deemed acceptable for the flexibility it affords. Further, since all end user interactions are intended to be through an IO API, having two different plugins is a small amount of extra work to provide a single interface that would operate on widely different scale deployments (e.g., one too small to have the IOD layer and one at the other extreme with a large IOD layer interfacing with a large, shared DAOS layer).

B. HDF5 to FFSIO Mapping

Since HDF5 has traditionally offered an interface focused on files and the internal data types, such as datasets, these concepts must be mapped onto the proposed FFSIO data storage concepts. This mapping is shown in Table I

TABLE I. HDF5 Data Model to FFSIO Data Model Mapping

HDF5	FFSIO
file	container
dataset	array
group	key-value store
attribute(s)	key-value store

In Section V below, the FFSIO types are described in more detail.

IV. IO Forwarding Layer

The IO Forwarding layer offers a mechanism to reduce the concurrency impact of the massive process count fan in onto the storage stack. The current trend of using both MPI and a node-level threading library like OpenMP, CUDA, or OpenACC is addressing the same issue, but limited to handling the parallelism on a single node rather than multiple nodes. Projected extreme scale platforms will have far fewer storage stack end-points per compute process or even compute node in which to receive requests and data. By reducing the number of simultaneous requests, delays can be reduced. This has been demonstrated for the file open operation with Lustre [17] and to some degree for accessing the storage devices themselves [18]. The BlueGene platform incorporated dedicated hardware to perform this role. The proposed functionality for this layer, beyond managing the number of connections to the IOD layer,

is to implement function shipping from the compute nodes to the IO nodes.

For the basic HDF5 calls, this will work the same as how the Nessie staging [16] shifted the collective IO data rearrangement calls to a reduced number of processes. The prototype implementation will only support accessing functionality already deployed to the IO nodes through an RPC mechanism. This initial implementation will use Mercury [29] to access the remote functionality. For dynamically defined functions, a different system will be required leveraging something like C-on-demand [1] or some other dynamic deployment and compilation or an interpreter system.

V. IO Dispatcher Layer

Strictly speaking, the IO Dispatcher layer and included functionality, such as burst buffers, is optional. All of the functionality, such as function and analysis shipping, transaction management, and managing asynchronous data movement can be handled by other portions of the stack. For an extreme scale platform, the IOD layer will be an essential pressure relief valve for the underlying persistent storage layer. By making it optional, the proposed stack can be deployed more easily on smaller clusters or for those with more constrained budgets. For simplicity, the rest of this section will describe a full stack including all of the proposed IOD components.

The core idea for IOD is to provide a way to manage IO load that is separate from the compute nodes and the storage array. Communication intensive activities, such as data rearrangement, can be moved to the IOD layer reducing the number of participants and message count. IOD has three main purposes. First, the burst buffers work as a fast cache absorbing write operations that then trickles out to the central storage array. It can also be used to retrieve objects from the central storage array for more efficient read operations and offers data filtering to make client reads more efficient. Second, it offers the transaction mechanism for controlling data set visibility and to manage faults that could expose an incomplete or corrupt data set to users. These transactions are local to the IOD layer until persisted to the DAOS layer eliminating the need for burdening the persistent storage with transient data. Third, data processing operations can be placed in the IOD. These operations are intended to offer functionality like data rearrangement and filtering prior to data reaching the central storage array.

While these ideas are not necessarily new, they are new twists on best of class efforts for these technologies. For example, offloading the collective two-phase data sieving from the compute nodes to reorganize data has proven effective at reducing the total time for writing data due to fewer participants involved in the communication patterns [16]. Beyond these broad items, there are many important details some of which are examined in more detail below.

A. FFSIO Data Model Types

With the shift from a directories and stream-of-bytes files model to the container and object model, some description is required to better understand how these concepts are being used as well as the raw benefits.

1) Container: As mentioned above, the concept of a container is similar to that of a file in a traditional file system. However, rather than being in a directory structure, each container essentially is stored in a hash-space allowing direct access to any container without regard to the current organizational context of the file system. For example, there is no need to navigate a directory hierarchy to name a particular container.

Functionally, a container plays the same role as a file in that it holds a collection of presumably related data intended to be accessed and manipulated as a unit. Since this is extended from HDF5 files, the container could also be viewed as a directory tree of objects where each directory entry specifies either a sub-directory (group) or some data or attribute. For the IOD layer, it is a collection of objects. For HDF5, interpreting the objects builds the structure.

2) Key-Value Store: This is the base type for the container. Since the container represents something akin to HDF5 files, everything is stored within a hierarchical namespace. The root namespace is represented by the base key-value store and contains a list of all of the objects for this portion of the namespace as well as additional key-value objects representing sub-groups for the hierarchy. Each of those key-value store objects works identically. Attributes are stored in a key-value store object, but use the multi-dimensional array and blob objects to store the values for the attributes.

3) Multi-Dimensional Array: By treating arrays as a special case separate from blobs, additional opportunities are enabled. For example, by knowing that an object is an array, proper slicing of that array onto IO nodes can be done without involving higher levels of the IO stack.

4) Blob: All other data is stored as a stream-of-bytes without regard to the actual data type.

B. Multi-Dimensional Array Data Distribution

For both IO performance and to aid in analysis and other data processing, the multi-dimensional array object can be split across multiple IO nodes. Each piece of this array is called a *shard*. The idea of sharding is to store a logically complete portion of a data set on a single storage target. This is similar in concept to the HDF5 hyperslab. The FFSIO stack supports sharding the data in the default or some other structured way as well as "re-sharding" based on application needs. For example, reordering the data so that a different dimension is the "fast" dimension may greatly improve the performance of a subsequent data analytics task. A common scenario where this is common is a Fortran code (column-major) writes data for a C code (row-major) to analyze. The IOD API supports the following sharding strategies:

- **contiguous**. fixed chunking, distributed in a round-robin fashion across the IO nodes.
- **chunked**. same as above but with irregular (sparse) chunking.
- **user-defined**. either contiguous or chunked, but user specifies where to place each individual shard.

It is possible to request the transformation of an object's physical layout to other formats resulting in multiple copies of the same objects in multiple formats if desired. Also, the user can pre-fetch objects from the storage cluster into the IO nodes or read them directly from the storage cluster. At the semantic level (HDF5), indices can be created for datasets resulting in being able to read through an index instead of directly from the base array.

All of these distinct alternatives result in having many different ways for executing the same analysis task. In the

subsequent discussions, we consider only data-movement optimization, i.e., sending the analysis code as close as possible to the data. In practice, this means we focus on identifying sharding of datasets and execute code accordingly over the appropriate shards.

C. IO Nodes

IOD processes are hosted on the IO nodes that interface a general compute area with the storage array. The IO nodes handle requests forwarded by the scientific applications, potentially integrate a tier of solid-state devices to absorb the burst of random or high volume operations, and organize/re-format the data so that transfers to/from the staging area from/to the traditional parallel file system can be done more efficiently. It also has the capacity to execute analysis on data recently generated by simulation applications running at the compute nodes, but not persisted to the storage array. As the data arrives, re-organization and data preparation can be applied in order to anticipate the execution of analytical tasks.

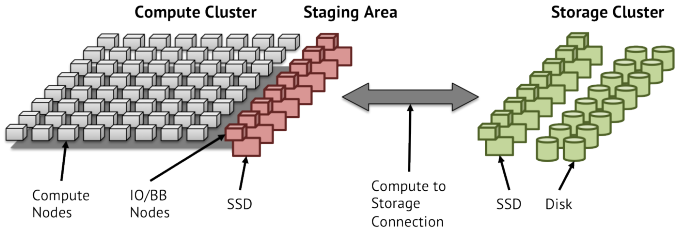


Fig. 3. Extreme Scale Architecture

A common configuration for this type of deployment is shown in Figure 3. The designated IO nodes (IONs) are connected to the compute nodes (CNs) through the same fast fabric (e.g., InfiniBand) while the connection to the external storage cluster is through a secondary, slower channel (e.g., 10Gb Ethernet). By providing additional storage on the IO nodes, such as SSDs, these nodes are capable of better regulating the IO pressure on the underlying storage array better than simple forwarding gateways. For this project, using something like SSDs on the IO nodes is termed a *Burst Buffer* and is discussed below.

D. Burst Buffers

The idea of burst buffers were initially explored in the context of data staging [3], [2], [21], [35]. These initial designs all use extra compute nodes to represent the data storage buffer given the lack of any dedicated hardware support for this functionality. The desired outcome of these initial studies is to motivate how such functionality might be incorporated and the potential benefits. Later, these concepts were proposed to be incorporated into the existing IO stack architecture [22], [5], [4].

In the case of the written IOD design, it describes a fixed-sized staging area that is partitioned on a per-application basis. As part of an application being deployed into the platform, each application will be allocated a fixed number of IO nodes for exclusive use during the application run. This provides guarantees about how much burst buffer space and processing capability will be available for the applications.

Future work will generalize this model to potentially support dynamic IO node allocation and examine the possibility of oversubscription. It will be strictly necessary to consider shared IO nodes for cases where the number of deployed applications exceeds the number of IO nodes. This first phase focuses on extreme scale application runs that use the vast majority of a platform rather than a capacity cluster where end-to-end performance is a lesser concern.

E. Data Versioning

Since space is limited in expensive, in compute area storage resources, a copy-on-write approach is used for new versions of the same data. For example, for a checkpoint/restart file, multiple versions will be written. The only parts of this container that must be replicated are those that have changed since the last write. With potentially many transactions written to the IOD layer because it is fast, this approach will enable additional output to be stored while reducing the space overhead. The inherent dependencies this introduces into the data are a lesser issue for the generally transient data in the IOD layer. For data intended for persistence in the DAOS layer, it may expose all versions of the data to corruption unnecessarily. This is explored in more detail in the Section VI.

F. Design Philosophy

The burst buffers design, as presented in the IOD documents, limits the placement of the function operators and SSD buffers to the IO nodes. The limitations of this design are acknowledged and the intent is to ultimately spread the IOD layer from the IO nodes into the compute area as well. This is intended to help address the limitations of the IO bandwidth and compute capability of these few nodes for data processing and also to take advantage of new layers in the storage hierarchy. By incorporating NVRAM into compute nodes, new options for buffering data prior to being moved to centralized storage become available and addresses potential concerns about SSD performance. For example, including a small amount of Phase Change memory into many or most compute nodes offers a way to move data outside of both the compute and IO path for data and communication intensive operations. Other projects [35] have shown this will have value, but the cost will have to be considered as part of the overall platform budget. This lessens the impact of some operators while offering additional options for places to store data.

Burst buffers being optional is a high level goal, but not considered in detail within the phase one design. If there is no burst buffer, all of the advanced functionality proposed for the IOD layer would have to work against the DAOS layer instead. For example, function shipping assumes it will operate on fast, local data within the IOD layer rather than against the globally shared DAOS layer that will likely still be disk for at least a couple more generations of platforms. With the additional desire to support using compute node resources for these operations, serious work will be required to make a fully functional end-to-end IOD layer implementation for a production system.

VI. DAOS Layer

The Distributed Asynchronous Object Storage layer serves as the traditional parallel file system interface layer for the storage devices. This is the consistent, global view of the underlying devices represented in this stack by the VOSD layer.

This is the layer where the container/object model is translated into the physical storage requirements dictated by the physical storage underneath (the VOSD layer). The two

key design elements of this layer are the handling of epochs and the mapping of containers and objects to the underlying storage.

There is a bit of a terminology shift between the IOD layer and the DAOS layer. For the IOD layer, a shard represents a portion of an object that is spread across potentially multiple IO nodes. For the DAOS layer, a shard represents the portion of a container that is spread across potentially multiple physical storage devices. The physical storage devices are represented by the VOSD layer described in Section VII.

While transactions at the HDF5 and IOD layer use the same term, at the DAOS layer the terminology shifts. Instead of transactions, the term *epochs* is used instead. Rather than attempting to introduce confusion, this is intended to help clarify how these concepts are used at different layers of the FFSIO stack. In the HDF5 and IOD layer, every operation has a transaction that may or may not ultimately be persisted to the DAOS layer. When a transaction is persisted to DAOS, it is termed an epoch to reflect that this is a persistent version of the container. For simplicity the epoch ID is the same as the transaction ID that was persisted. Unlike transaction IDs, epoch IDs generally are not consecutive reflecting that not all transactions will be persisted to DAOS.

To deal with the potentially missing data versions between epochs because not all transactions are persisted, a special procedure must be followed. The "flattening" process combines multiple copy-on-write versions of a transaction into a single epoch. Since this stack uses a copy-on-write approach to reduce the space requirement for new versions of existing files all of the changes between the last epoch and the current epoch must be combined into a single entry. While not a cost free operation, it is generally considered inexpensive since a backwards combining of transaction blocks can be made ignoring any block that is already part of the combined changes.

The current implementation has the DAOS layer map the container/object data model onto a directory/file data model used for most existing file systems. Should a fully object-based file system be deployed at the VOSD layer, this mapping would be unnecessary. The current projections suggest that a standard POSIX-like file system will likely be used at the lowest level on each storage device requiring the mapping at some level. To perform this mapping, DAOS considers the following.

Each container is represented by a directory on some storage device containing symbolic links to all of the shards it contains and maintains the epoch ID. In particular the Highest Committed Epoch is an important concept for quickly identifying which version of a shard to retrieve and to block writes to older epochs since those have been committed.

Overall, the DAOS layer serves as the shared persistent storage interface for the IO stack. In the case of a data center-wide storage array, the DAOS layer would be shared across all of the platforms with the upper layers being local to each individual platform.

To address consistency issues between platforms, containers at the DAOS layer must know of every transaction. To address this, a container is updated every time a new transaction is created for it and closed or aborted. This ensures that if multiple platforms are writing to the same container sequentially that they will not have conflicts in the highest transaction number. The FFSIO stack does not support multiple applications from the same or different platforms using a shared DAOS layer to write to the same container at the same time. This functionality is not supported by popular existing paralell file systems either.

A. Design Philosophy

The DAOS layer is the key storage management layer for this system. By handling the translation between user-level concepts and the underlying hardware, performance and functionality are both important. The choice of an object interface is influenced by the performance gains achieved by the data analytics community for non-shared data access. With the system design favoring requiring this operation mode, using an object interface fits naturally. With the broad array of object-based storage devices hitting the market, this layer may thin outsourcing much of the object creation and management to these speciailzed devices.

Since this is the layer at which a storage system will be shared by multiple platforms, consistency is also a concern. By shifting to an object model and moving away from a POSIX-style directory tree, maintaining consistency will be easier. No longer will a consistent view of a particular set of files (containers) be required. Instead, only a single container need be consistent. With container sharing between platforms generally being limited to downstream analysis routines, waiting for a new epoch to be persisted can serve as an analysis trigger.

Issues related to the handling of transactions and epochs are discussed in Section VIII-A. Maintaining storage system scalability with this functionality will be challenging.

VII. VOSD Layer

The Versioning Object Storage Device (OSD) layer operates as the interface for each persistent storage device used to support the parallel storage array. In the purest form, it uses a local file system to arrange storage of objects that represent parts of the higher level objects in containers.

The base level implementation continues the space optimization of only storing changes for new versions by using a copy-on-write file system. The prototype uses ZFS [34] for the known stability and integration with Lustre. In a production version of the FFSIO stack, btrfs [26], The Linux B-Tree File System, given its open-source backing and GPL licensing, is a likely long-term choice.

At a more detailed level, the design for VOSD is an increment beyond the current Lustre Object Storage Device design to incorporate the idea of shards and the versioning aspects of transactions/epochs. For every DAOS shard, the VOSD has information for storing and accessing the currently committed version, the Highest Committed Epoch, as well as a staging dataset representing the next version of the object being stored. Both of these are combined in a shard root.

For data integrity, an intent log is maintained as part of the underlying file system enabling fault recovery.

Beyond the functionality to incorporate and expose the copy-on-write nature of the underlying file system and the semantics for storing and processing shards and their associated epochs, this is largely an evolution of the existing Lustre OSD layer.

VIII. Broader Design

Several concepts crosscut many of these layers and are best described in a single location. For example, transactions and epochs are visible from the user API level down into the VOSD

layer. While each layer affects the concept, it is best to look at each concept across all of the layers.

In the subsections that follow, we examine transactions and epochs, metadata management, function and analysis shipping, and the arbitrary connected graphs support.

A. Transactions and Epochs

As mentioned above, the transaction mechanism manifests in two forms. From the user level down through the IOD layer, they are called transactions and are used to judge whether or not a set of distributed, asynchronous modifications across a set of related objects (i.e., within a container) is complete or not. It is also used to control access by treating the transaction ID of committed transaction as a version identifier. At the DAOS layer and below, they are called epochs and represent persisted (durable) transactions from the IOD layer. Each of these offers different functionality, but are connected as is explained below.

1) Transactions: To understand how transactions are used in the IOD layer, some terminology and concepts must be explained first. At the coarsest grain level is a container. Each container provides the single access context through which to access a collection of objects. Transactions are the way that a series of modifications to the objects within a container are treated atomically. Conceptually, containers correspond to a something akin to an HDF5 file in a traditional file system. The objects in each container represent different data within a file. The three initially defined object types are key-value stores, multi-dimensional arrays, and blobs. The easiest way to understand these types is to evaluate these from the perspective of an HDF5 file, the initial user interface layer. The key-value store represents a collection of attributes or groups. The array represents a potentially multi-dimensional array. The blob represents a byte stream of arbitrary contents. The fundamental difference between an array and a blob is that the array has metadata specifying the dimension(s). At the physical layer within the IO nodes, all of these objects may be striped across multiple IO nodes. Given this context, the transactions come in two forms.

First is a single leader transaction where the IOD manages based on calls from a single client. The underlying assumption is that the client side will manage the transactional operations itself and the single client is capable of and responsible for reporting to the IOD how to evolve the transaction state.

The second form is called multi-leader and has the IOD layer manage the transactions. In this case, when the transaction is created, a count of clients is provided to the IOD layer. As clients commit their changes to the container, the reference count is reduced. Once the count reaches 0, the transaction is automatically committed.

2) Epochs: The Epoch mechanism differs from transactions. Instead of focusing on when a particular output is complete, an epoch represents incremental persisted container copies. To simplify the mapping between an IOD transaction and the DAOS epochs, when an IOD transaction is persisted to DAOS, the IOD transaction ID is the used as the epoch ID. The key difference is that at the DAOS layer, some transaction (epoch) IDs will not be represented with data since not all IOD transactions are necessarily persisted. Maintaining this ID continuity is critical for multiplatform use. Since the shared point is the DAOS layer, any user adding a new version to a file must be able to determine the most recent transaction ID no matter from where the container was updated last.

3) Design Philosophy: Undocumented, but inherent in the design of these transactions is how faults are detected. The initial design assumes the current Lustre fault detection mechanism that can determine if a process or node is no longer reachable. This detection happens at the DAOS layer and when a fault is detected, the rollback process is pushed up to the IOD layer for all non-persisted or non-committed transactions. This defines how a fault will be detected and what will trigger a passive fault recovery (i.e., transaction abort). The challenge with this approach will be scalability. Existing Lustre systems can use the IO node status as a proxy for compute area status. Since the DAOS layer must now know the state of every node, if not every process on every node, to properly handle transactions, some scalable status tracking mechanism is required.

There are two steps for beginning a transaction on a container. The first step is for one or more process to open the container. This handle can be shared eliminating the need for every participating process to hit the IOD layer to open the file. The second step is a call to determine how many leaders will participate in the transaction. In the single leader case, there is no IOD-side aggregation of success/fail statuses to determine the final transaction state. Instead, it is assumed that the client will fully manage the transaction. In the multi-leader model, some subset from 2 to n where n is the count of all processes, declare themselves a leader for this container operation to the IOD layer. Any number of processes can participate in modifying container without regard to whether or not they are a leader. Once each leader has finished, with the assumption that any clients a leader may be responsible for are finished as well, the IOD layer aggregates those responses to either commit or abort the transaction. For scalability and performance, phase two is favoring single leader transactions. With libraries like D^2T [15], [13], [14] to ease implementing client-side transactions, this burden is lessened.

Ultimately, with the passive detection of faults for transaction leaders, the transaction mechanism can work very well. A mostly unstated restriction that is being relaxed for phase two is that every sequential transaction on a container is considered dependent on the earlier transaction. Should one output be delayed and the subsequent five succeed, when the delayed process finally fails, all six transactions are rolled back. The thought of using this mechanism to store subsequent checkpoint outputs in the same container to both save space, but not care if one fails, cannot work in the current form. This has been acknowledged and is being relaxed requiring a new parameter to the creation of a transaction determining if it will be dependent or not. The downside to supporting this functionality is the reduced ability to use copy-on-write to reduce space pressures. If a transaction is allowed to fail and not affect subsequent transactions, the beginning state for subsequent transactions must the committed version prior to the current transaction. For example, if transaction ID 5 is marked as independent and 4 was previously committed, then transaction ID 6 will have to use version 4 as the base for copy-on-write. A more in depth discussion of how transactions and checkpoint restart work is presented in BAD Check [6].

B. Metadata Management

Metadata management has been a perennial challenge for parallel storage systems. Eliminating metadata management as a special case and instead treating it just as data is a central

design goal of the Fast Forward project. This is a hybrid approach to metadata management that is halfway between providing no inherent metadata support and having a fully integrated, but separate metadata management system.

Eliminating metadata as a core component of a file system is not new. It has been explored as part of the Light Weight File Systems project [23]. In LWFS, the metadata service is explicitly limited to a user task with the storage layer limited to data storage/retrieval, authorization, and authentication. This approach proved workable. Using this hybrid approach is less common [33] and introduces other issues.

IOD and DAOS both share a philosophy that they will have to maintain the metadata about how the physical pieces of the logical objects are striped and where they are placed. The primary metadata management is done at the DAOS layer with the IOD layer relying on the DAOS layer for all authoritative information about containers and objects. The only place where the IOD layer manages metadata for itself is to manage how the different objects are striped across the IO nodes.

1) Design Philosophy: While the metadata design is not fully defined, there are a few things that are intended. For example, there will be a standard, well-known container that is the system metadata. This includes the list of all other containers. This container is treated like any other data in the system and striped as appropriate. Unfortunately, this still couples the metadata to a single object that must serialize access. If the metadata, including information about striping and other data layout operations were separated completely from the data path, more scalable throughput could be achieved. The real challenge of this is introduced by the IOD, DAOS, and VOSD layers collectively. Each of these requires some different metadata storage and the migration is invisible to the user. Supporting fully independent metadata with this model is difficult. Serious thought on how to do this effectively outside the data path will be considered for phase two.

C. Function and Analysis Shipping

A client/server architecture is implemented for the Compute Node-IO Node communication model. Every ION runs an IOFSL (IO Function Shipping Layer) server. The IOFSL client is integrated into the HDF5 library running on each CN. A client can forward requests to any number of IONs. Every IO operation issued by HDF5 is asynchronously shipped to the IOFSL server and asynchronously executed. As it is currently implemented, the only functionality that can be "shipped" already exists on the IONs and is activated using RPC calls. This will be re-evaluated for phase two to provide more dynamic functionality.

1) Design Philosophy: The advantages of in transit and in situ processing is well documented. For example, PreDatA [35] evaluates different operator placement decisions and when each is advantageous. Inherent in this approach is the need to ship arbitrary code to run in various locations. Security considerations aside, something like C-on-demand [1] or some other dynamic code generation and execution system will be required. Lightweight containers [20] could also be used, but with potentially higher management overheads.

D. Arbitrarily Connected Graphs

What people popularly consider Big Data applications fall into two broad categories. First, data processing tasks that

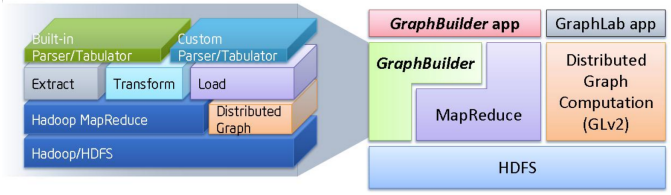


Fig. 4. GraphLab and GraphBuilder stacks

can fit into the MapReduce model where data is tagged and sorted to discover relationships. These sorts of applications only require scale out rather than scale up. Scaling out requires replicas to process data simultaneously, but do not need to coordinate for that data processing. Scaling up, what scientific simulations do, requires sometimes serious coordination between the processes for any of them to succeed. In the middle are graph applications that, with some replicated data, can be made to fit reasonably well into the MapReduce model. The challenge is having access to the edge and vertex lists effectively to build partitions for independent processing. GraphBuilder [12] is a tool to generate effective graph partitions reducing the load for using MapReduce to process graph data sets. GraphLab [19] offers a way to process these graphs efficiently for parallel platforms using a minimum of communication. Using these tools as motivators, changes to the HDF5 interface and the underlying storage infrastructure is proposed. The following illustrates the architecture of both frameworks:

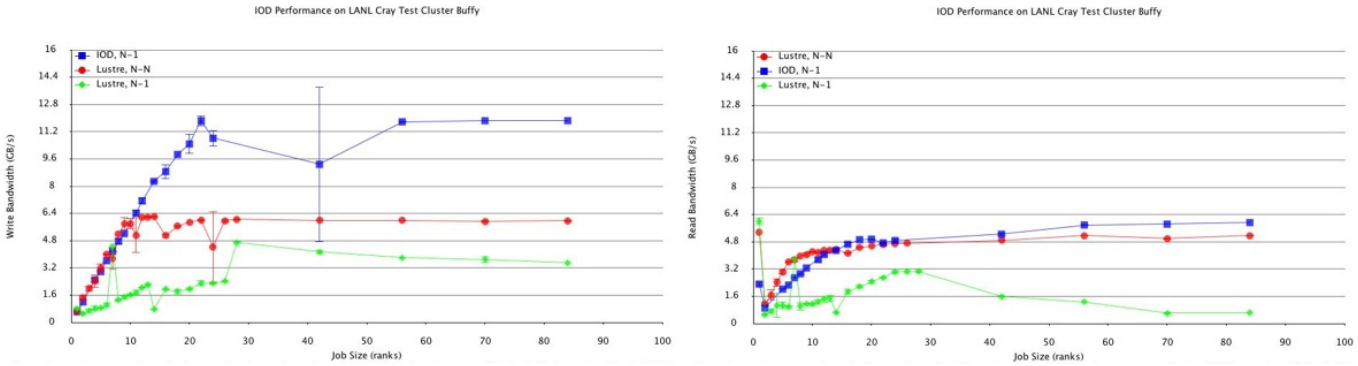
In order to make both of these tools work on top of the extreme scale stack, they both have to be modified. After these modifications are implemented, GraphBuilder will be able to write the partitioned graph in the newly proposed HDF5 files which will thus be stored in the IOD nodes (or IONs) in a parallel-optimized way. On the GraphLab side, HDF5-awareness will allow the library to perform at high speeds by benefiting from the new features, such as the function shipping. In general both frameworks will be modified so that calls to HDFS-based formats are replaced by the proposed HDF5 equivalents. This is referred to as the HDF Adaptation Layer or HAL and will provide, from the GraphBuilder/GraphLab point of view:

- capability for storing the newly proposed HDF5 format
- association of network information to vertices/edges
- shipping computation to the IONs
- asynchronous vertex updates
- efficient data sharing among CNs
- computation over versioned datasets

The initial phase of this project has determined the necessary changes in the HDF5 format to support these features. These identified features will be proven during phase two with a demonstration of GraphBuilder and GraphLab.

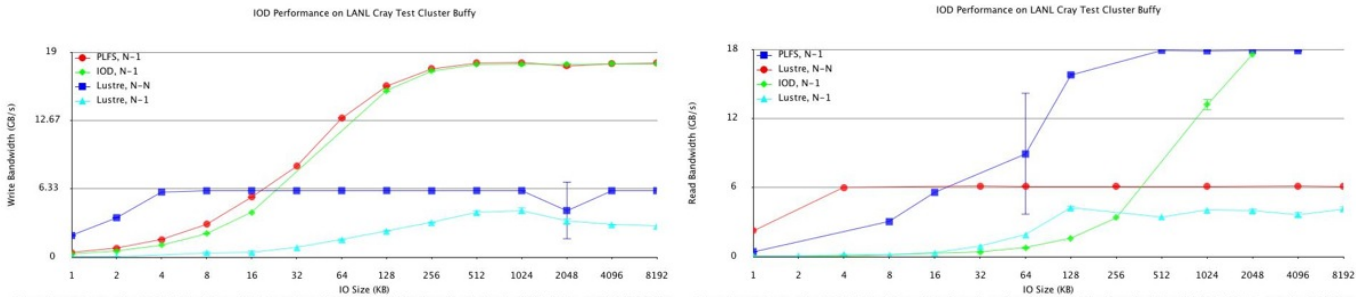
IX. Demonstration

This stack has an early prototype implementation intended to test concepts rather than performance and scalability. It has focused on examining the interaction of the different APIs for each layer to flesh out any detailed requirements or concerns that may have been missed in the conceptualization of this IO stack. To demonstrate the viability of the IO stack described



(a) Write Hosts (b) Read Hosts

Fig. 5. Functionality Demonstration Validation for Number of Hosts



(a) Write Size (b) Read Size

Fig. 6. Functionality Demonstration Validation for Data Sizes

in this paper, we show some very early performance results from the untuned prototype.

All of the tests are performed on a blinded for review machine. It consists of 64 compute nodes each with dual, 8 core Intel Xeon ES-2670 CPUs at 2.6 GHz and 64 GB RAM. The interconnect is Cray Aries. There are 14 IO nodes consisting of single socket, 8 core Intel Xeon ES-2670 CPUs at 2.6 GHz with 32 GB of RAM. There are also a metadata, login, and 2 boot nodes. The storage array has disk and SSD partitions. One is a DDN Lustre system with 192 TB disk usable with a minimum of 5 GB/sec performance. The rest are SSDs consisting of an EMC flash array connected via FDR InfiniBand with 22 TB and 48 GB/sec write performance.

We run two different sets of tests. The first set in Figure 5 show the reading and writing performance for different numbers of hosts. Each read or write is 4 GB against the x-axis number of hosts. The second set in Figure 6 show the performance of reading and writing different sizes for 56 clients, the smallest client count when performance stabilizes in the number of hosts tests. The performance of both of these tests are reported to give a very rough idea of the overhead that might be involved. Rather than a true overhead, this should be considered the maximum overhead that should be expected once an optimized, fully functional IO stack is deployed without relying on translating to an underlying parallel file system.

X. Conclusions

The Fast Forward Storage and IO Stack project has designed a good first pass at addressing the requirements for an extreme scale data storage mechanism. By preferring a high level user API like HDF5 rather than using the POSIX interface, more advanced functionality can be incorporated with less end-user impact. The introduction of the IOD layer with buffering will absorb the difference between the compute node IO demands and the available bandwidth in the storage array. With DAOS supporting translating the container and object model to the underlying storage options, different storage technologies can be deployed over time.

With the overall stack design a prototype implementation complete, refinements, such as fault detection and recovery, can be designed and tested. These and other activities for phase 2 will ultimately generate what is likely to be the next generation storage stack for extreme scale platforms.

XI. Acknowledgements

blinded for review

References

[1] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky. Just in time: adding value to the io pipelines of high performance applications with jitstaging. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 27–36. ACM, 2011.

[2] H. Abbasi, J. Lofstead, F. Zheng, S. Klasky, K. Schwan, and M. Wolf. Extending i/o through high performance data services. In *Cluster Computing*, Luoisiana, LA, September 2009. IEEE International.

[3] H. Abbasi, M. Wolf, and K. Schwan. LIVE data workspace: A flexible, dynamic and extensible platform for petascale applications. In *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 341–348, Washington, DC, USA, 2007. IEEE Computer Society.

[4] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring. Jitter-free co-processing on a prototype exascale storage stack. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5, April 2012.

[5] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. Storage challenges at los alamos national lab. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5, April 2012.

[6] J. Bent, B. Settlemyer, H. Bao, S. Faibish, J. Sauer, and J. Zhang. Bad check: Bulk asynchronous distributed checkpointing and io. In *Proceedings of Tenth Parallel Data Storage Workshop at Supercomputing 2015*, 2015.

[7] P. J. Braam. The lustre storage architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre, Nov. 2002. http://www.lustre.org/docs/lustre.pdf.

[8] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, Oct. 2000. USENIX Association.

[9] M. L. Curry, L. Ward, and G. Danielson. Motivation and design of the sirocco storage system, version 1.0. Technical report, Sandia National Laboratories, Albuquerque, New Mexico, 2015. http://www.cs.sandia.gov/Scalable_IO/sirocco.

[10] Fastforward storage and i/o stack design documents. Intel FastForward Wiki, February 2014. https://wiki.hpdd.intel.com/display/PUB/Fast+Forward+Storage+and+IO+Program+Documents.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the NineteenthACM Symposium on Operating Systems Principles*, pages 96–108, Bolton Landing, NY, Oct. 2003. ACM Press.

[12] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: Scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.

[13] J. Lofstead, J. Dayal, I. Jimenez, and C. Maltzahn. Efficient transactions for parallel data movement. In *The Petascale Data Storage Workshop at Supercomputing*, Denver, CO, November 2013.

[14] J. Lofstead, J. Dayal, I. Jimenez, and C. Maltzahn. Efficient, failure resilient transactions for parallel and distributed computing. In *Data Intensive Scalable Computing Systems (DISCS), 2014 International Workshop on*, pages 17–24, Nov 2014.

[15] J. Lofstead, J. Dayal, K. Schwan, and R. Oldfield. D2t: Doubly distributed transactions for high performance and distributed computing. In *IEEE Cluster Conference*, Beijing, China, September 2012.

[16] J. Lofstead, R. Oldfiend, T. Kordenbrock, and C. Reiss. Extending scalability of collective io through nessie and staging. In *The Petascale Data Storage Workshop at Supercomputing*, Seattle, WA, November 2011.

[17] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.

[18] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing variability in the IO performance of petascale storage systems. In *Proceedings ofSC2010: High Performance Networking and Computing*, Nov. 2010.

[19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[20] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[21] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[22] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield. Zest checkpoint storage system for large supercomputers. In *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, pages 1 –5, nov. 2008.

[23] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener. Lightweight I/O for scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sept. 2006.

[24] Object-based storage architecture: Defining a new generation of storage systems built on distributed, intelligent storage devices. Panasas Inc. white paper, version 1.0, Oct. 2003. http://www.panasas.com/docs/.

[25] B. Powlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementations. pages 137–152, 1994.

[26] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[27] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the USENIX FAST '02Conference on File and Storage Technologies*, pages 231–244, Monterey, CA, Jan. 2002. USENIX Association.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[29] J. Soumagne, D. Kimpe, J. A. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. B. Ross. Mercury: Enabling remote procedure call for high-performance computing. In *CLUSTER*, pages 1–8. IEEE, 2013.

[30] The HDF Group. Hierarchical data format version 5, 2000-2014. http://www.hdfgroup.org/HDF5.

[31] F. Wang, M. Nelson, S. Oral, S. Atchley, S. Weil, B. W. Settlemyer, B. Caldwell, and J. Hill. Performance and scalability evaluation of the ceph parallel file system. In *Proceedings of the 8th Parallel Data Storage Workshop*, pages 14–19. ACM, 2013.

[32] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 2006Symposium on Operating Systems Design and Implementation*, pages 307–320. University of California, Santa Cruz, 2006.

[33] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI'06*, Seattle, WA, Nov. 2006.

[34] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In R. C. Burns and K. Keeton, editors, *FAST*, pages 29–42. USENIX, 2010.

[35] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA - preparatory data analytics on Peta-Scale machines. In *In Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium, April, Atlanta, Georgia*, 2010.