Date: 25 June 2014	# EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT # FAST FORWARD STORAGE AND I/O # MILESTONE: 8.5 – Final Report

LLNS Subcontract No.	B599860
Subcontractor Name	Intel Federal LLC
Subcontractor Address	2200 Mission College Blvd. Santa Clara, CA 95052

Table of Contents

Contents

Revision History

Date	Revision	Description	Authors
25 June 2014	1.0	Final Report submitted to stakeholders	Intel The HDF Group EMC Cray

1 Executive Summary

Although HPC storage architecture has remained remarkably consistent from the terascale to the petascale eras, three emerging trends have rendered it infeasible for exascale. The first trend, due to the physics and economics of physical storage, is towards a new tiered storage architecture in which solid state storage closely coupled to the HPC cluster fabric supports ever increasing performance requirements measured both in terms of IOPs and bandwidth of HPC workflows. The second trend, driven by improvements in computational power, is towards increasing data volume and metadata complexity. The third trend, due to the need to limit power consumption while continuing to scale computational performance, is towards ever increasing core and node counts which requires matching scaling of application concurrency and directly increases the frequency of hardware failure.

These trends stretch the familiar POSIX I/O stack beyond its limits in two ways. Firstly, its fundamental storage abstraction, the POSIX file, imposes unscalable consistency requirements on concurrent accesses. This forces application and middleware developers to abandon the manageability of the shared file I/O model in favor of file-per-process. This pollutes the filesystem namespace with what is effectively application metadata and also moves rather than solves the scaling problem. Secondly, POSIX has no transactional semantic to allow the update, with guaranteed integrity in the presence of either application or system failure, of large and complex datasets. This forces applications to create new files at every stage of the workflow to ensure existing data will not be compromised on failure, further polluting the namespace.

This report describes the challenges imposed by these trends in more depth and presents the design and prototype of a complete I/O architecture that addresses them: the Exascale Fast Forward (EFF) I/O stack. The top of the stack features a new version of HDF5, extended to support the new EFF storage semantics for high-level data models, their properties and relationships. In the middle of the stack is IOD, the I/O Dispatcher, which stages data between storage tiers and optimizes placement. Finally at the bottom of the stack is DAOS, Distributed Application Object Storage, which provides scalable, transactional object storage containers to encapsulate entire exascale datasets and their metadata.

All layers in the stack share common features which are essential to enable scalable and fault tolerant storage access for exascale HPC and big data. The major departure from POSIX is to support transactional multi-version concurrency control. This not only eliminates traditional sources of serialization for increased scalability and workflow concurrency, but also guarantees the consistency of complex distributed data and metadata in the presence of arbitrary hardware and application failure. This in turn enables important resilience features such as the data integrity checks in the HDF5 and IOD layers that detect silent data corruption. All operations are asynchronous to allow applications to overlap I/O with computation and enable the storage stack to exploit opportunities to aggregate and reschedule.

Each stack layer also provides unique functionality and expands on the layer below it. For example, DAOS provides distributed transactions and scalable object I/O. IOD adds key-value and multi-dimensional array object semantics while HDF5 adds map objects and supports analysis shipping for searching and indexing data. The value of the stack for HPC is demonstrated through a version of the VPIC-IO benchmark modified to use the new HDF5 and a new storage API for graph analytics, also layered above HDF5, demonstrates the stack for Big Data workloads.

The EFF I/O stack, by guaranteeing read consistency in the presence of concurrent writers, supports I/O not only for single applications but also more complex runtimes that integrate multiple applications via parallel data pipelines. The application modifying a dataset, such as a simulation adding a new timestep, does not have to block for readers, which in turn are guaranteed to operate on consistent snapshots of the data they access. This facilitates and simplifies consumer-producer workflows where data produced on the compute nodes of an HPC cluster is analyzed by other applications running concurrently on burst buffer or storage nodes.

The remainder of this document discusses the EFF I/O stack and how it addresses the challenges of exascale in more depth. Section 3 discusses the design of the overall stack and each of its layers, Section 4 describes the implementation of the major features and Section 5 presents technical findings discovered during prototype implementation and evaluation. Lessons learned co-developing multiple layers of integrated software distributed both across geography and multiple institutions and recommendations for future projects are described in Section 6. Finally, Section 7 describes areas for further research and development to transform the prototype system into a production solution suitable for exascale.

2 Terminology

Throughout the remainder of this document, the following abbreviations will be used.

- **ACG** - Arbitrarily Connected Graph
- **BB** - Burst Buffer - the physical flash storage mounted on each ION
- **BSP** - Bulk Synchronous Processing
- **CN** - Compute Node
- **CV** - Container Version
- **DAOS** - Distributed Application Object Storage
- **DOI** - Data of Interest
- **EFF** - Exascale Fast Forward
- **ETL** - Extract, Transform, Load
- **FID** - Lustre unique object identifier
- **HAL** - HDF5 Adaptation Layer
- **HDF5** - Hierarchical Data Format, v5
- **HDFS** - Hadoop Distributed File System
- **HCE** - Highest Committed Epoch
- **IOD** - IO Dispatcher (software running on the ION)
- **ION** - IO Node
- **IOPS** - I/Os Per Second
- **KV** - Key-value object
- **LOC** - Lines Of Code
- **MDHIM** - Multi-dimensional Hashing Indexing Middleware (manages IOD KV objects)
- **MDS** - Lustre Metadata Server
- **OSC** - Lustre Object Storage Client
- **OSD** - Lustre Object Storage Device
- **OSS** - Lustre Object Storage Server
- **OST** - Lustre Object Storage Target
- **PGAS** - Partitioned Global Address Space
- **PLFS** - Parallel Log-structured file system (stores IOD blobs and arrays in BB)

- **RC** - Read Context
- **SN** - Storage Node
- **VOL** - Virtual Object Layer
- **VOSD** - DAOS Versioning Object Storage Device

3 Design

This project anticipates major challenges for exascale storage in both software and hardware. The software problem is motivated by the expectation that the current storage software stack, based primarily on POSIX I/O and which, perhaps unexpectedly, survived the terascale to petascale transition, can no longer be relied upon for the exascale era. The hardware problem is motivated by the increase in failure rates due to the sheer size of systems expected at exascale and the economic trends of disk-based storage, which forecasts a future in which disk-based storage can provide capacity at reasonable cost but not performance.

Two main factors threaten the viability of POSIX I/O and the current storage software stack: increasing concurrency and larger and more complex metadata. Node counts are expected to approach the hundreds of thousands with potentially hundreds of MPI ranks on each node. This creates an extreme scalability requirement for the I/O system that cannot tolerate any unnecessary synchronization. Increasing metadata volume and complexity creates a need for high level object oriented I/O models that require I/O subsystems capable of supporting very diverse workloads. The POSIX file abstraction is poorly suited to these requirements and I/O middleware and applications based on POSIX are forced towards schemas that create separate files for each process and each type of usage, effectively polluting the POSIX namespace with application and middleware metadata and making it unmanageable. A new storage abstraction to replace POSIX file is therefore required that can encapsulate an entire exascale dataset, including all application and middleware metadata.

The vast number of hardware components anticipated in exascale systems will make both application and storage failure the norm. Coupled with an ever increasing volume of data, this renders unviable any recovery procedures that must scan the entire storage system. POSIX provides no means to guarantee atomic update in the presence of failure and current I/O middleware is forced to write updates to new files to avoid corrupting existing data with partial updates. Transactional storage APIs are therefore required to enable distributed applications to group their updates into consistent state changes applied across multiple storage systems. These systems will be required to automatically rollback incomplete transactions in the event of either application or storage failure. These APIs must support fully asynchronous operation to decouple processes collaborating on the same transaction and enable full overlap of computation and I/O.

The increasing probability of silent data corruption also requires new solutions. Supercomputers today, operating on data volumes orders of magnitude smaller than exascale, experience frequent incidents. Verification of data integrity end-to-end is therefore a requirement at exascale.

As improvements in disk capacity have completely outpaced improvements in performance, the minimum number of disks needed for performance has grown to be much larger than the number required for capacity. Unfortunately, the economics of disk makes purchasing disks for performance prohibitively expensive for HPC storage systems. Fortunately, flash media technologies have advanced to the point where they can provide affordable performance, if not affordable capacity. Exascale storage must therefore be a hybrid system in which a staging tier, termed a *burst buffer,* provides performance and offloads data to a slower disk tier that provides capacity.

This project addressed these issues with the creation of the EFF I/O stack, which has a layered architecture with three main APIs. At the top of the stack, a new version of HDF5 provides high level object storage. At the bottom of the stack, the DAOS API replaces the POSIX file with a transactional, shared-nothing distributed object store, the DAOS container. Situated between the two, IOD manages the burst buffer storage tier and matches the diverse I/O requirements of the layer above to the storage capabilities of the layer below.

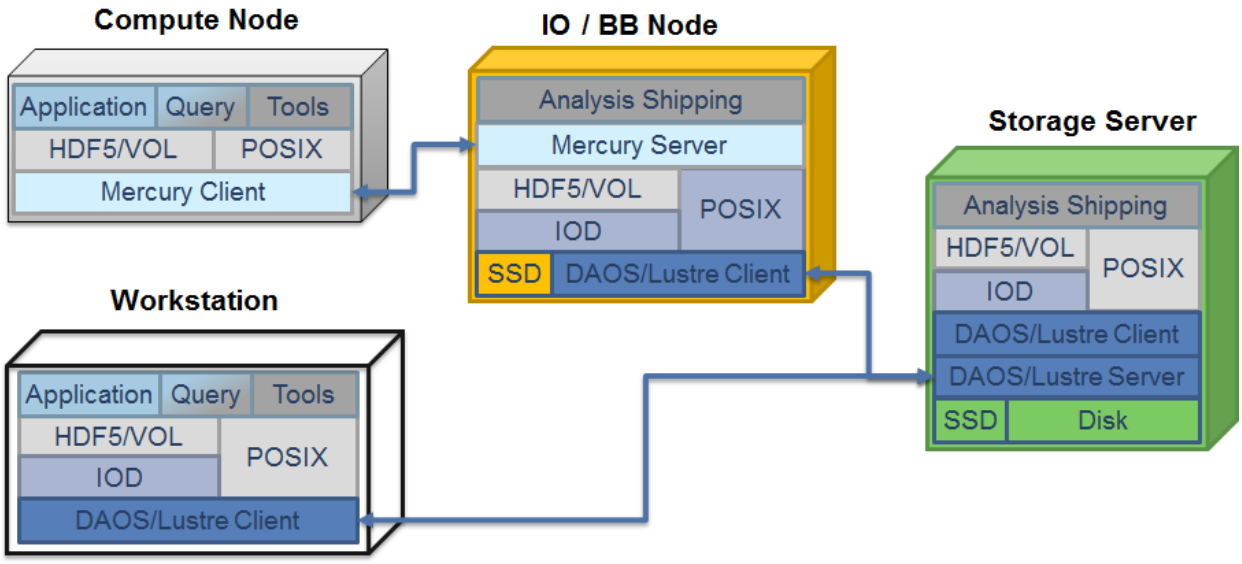

Figure 1: EFF Stack Configuration

Economics currently dictate that the flash storage tier should be consolidated on dedicated nodes; therefore the EFF stack co-locates the DAOS client with IOD on these dedicated nodes and connects application processes running HDF5 on compute nodes to IOD via the Mercury I/O forwarding service. Lustre, augmented to support the DAOS API continues to support POSIX as does Mercury, to enable conventional I/O libraries still to be used within EFF applications for legacy I/O.

The EFF stack supports distributed, asynchronous, transactional I/O in all three of its main APIs. By grouping a set of updates distributed both in space and time into a single atomic unit, EFF transactions atomically transition stored data from one consistent state to another. Furthermore, due to the multi-version concurrency control implementation of

these transactions, consumers of the data see consistent point-in-time snapshots even while another application is performing updates. This enables concurrent producer/consumer workflows such as simulation and analysis.

The EFF stack also introduces checksums as a parameter in all I/O operations in the HDF5 and IOD APIs. These verify that data and metadata integrity has been preserved from the application to persistent storage and back by detecting data corruption. Future development of, and integration with, underlying replication schemas made possible by the EFF transaction model will eventually enable recovery of corrupted data.

3.1 Stack Layer Abstractions

The EFF stack adopts objects as the basic units of organization and storage throughout the stack. Applications use HDF5 to both organize and access data and metadata using a rich set of HDF5 abstractions. IOD provides a simpler and more generic set of abstractions that HDF5 uses to represent the application-level data model and DAOS in turn provides an extremely simple object model to IOD.

Object primitives, at all levels of the stack, increase the opportunities for optimization based on natural access patterns and segregate data into "like pieces" rather than forcing potentially disparate data that happened to be written close in time but not in concept into close proximity on storage.

The layered architecture allows each layer to optimize its own use of the relevant storage hardware. Each layer is responsible for translating guidance expressed in terms of its own abstractions by the layer above into guidance for the layer below. For example, IOD exposes information on data locality in the burst buffer to allow analysis operations to be shipped by the HDF5 layer to the ION where the data is resident. The layered architecture also facilitated co-development by allowing experts at each layer to concentrate on their respective areas while contributing to the design of the common features.

Table 1 below shows how each layer represents its abstractions and metadata using the abstractions made available to it from its underlying layer.

Table 1: Abstractions and Metadata by Layer

HDF5 Abstraction	IOD Abstractions	DAOS Abstraction
H5File	Container	Container
H5Group	KV object for links to members KV object for H5 Metadata KV object to locate Attributes Array objects for Attributes	Set of DAOS objects distributed across DAOS container shards
H5Dataset	Array object Blob objects if var-length data KV object for H5 Metadata KV object to locate Attributes Array objects for Attributes	Set of DAOS objects distributed across DAOS container shards
H5Map	KV object Blob objects if var-length values KV object for H5 Metadata KV object to locate Attributes Array objects for Attributes	Set of DAOS objects distributed across DAOS container shards
H5NamedDatatype	Blob Object KV object for H5 Metadata KV object to locate Attributes Array objects for Attributes	Set of DAOS objects distributed across DAOS container shards
H5Datatype H5Dataspace H5Properties	KV pair in H5 Metadata KV	Data in a DAOS object

3.2 DAOS

The Distributed Application Object Storage (DAOS) API is an object-based I/O interface designed for scalability and resilience. Its container abstraction virtualizes object storage systems distributed across multiple servers to provide upper layers of the EFF stack with a private object address space to store complex data structures. DAOS supports multi-version concurrency at byte granularity to eliminate unintended serialization through false sharing conflicts and allows updates by multiple distributed processes to multiple distributed objects to be grouped into transactions to guarantee consistency in the face of application and system failures. More details on the DAOS API can be found in [D4].

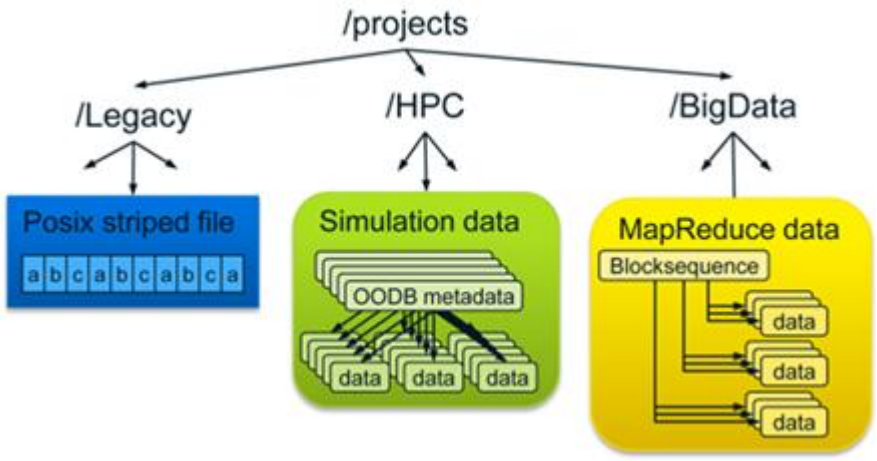

Figure 2: POSIX Namespace

DAOS containers are accessed via the POSIX namespace, which is used to confer ownership and enforce security. Each DAOS container within the namespace provides an object address space partitioned over multiple storage systems, where each object is a simple byte array. Each partition, called a container shard, is the unit of both concurrency and fault-tolerance. Higher levels in the I/O stack may therefore distribute their own data structures over DAOS objects in different container shards to achieve horizontal scalability. Furthermore, the consistency guarantees provided by transactions ensure that replication and erasure coding can be used to ensure durability and availability. DAOS also provides query APIs to allow its callers to determine the locality and topology of the underlying storage through its query APIs. This enables upper levels to determine optimal placement for both performance and resilience.

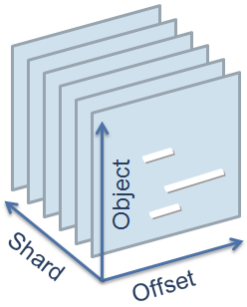

Figure 3: DAOS container address space

DAOS avoids scaling problems and overheads common to traditional POSIX-based file systems through careful attention to its I/O operations. These are limited to read, write and punch. Punch is the equivalent of writing zeroes, but also frees underlying storage and avoids the need for object create/destroy. These operations may be executed on arbitrary object extents with arbitrary alignments that are preserved over-the-wire to avoid the need for locking. All three operations are idempotent so that individual operations may be repeated until successful or abandoned.

Remaining container operations, including open and commit are executed by a process leader rather than by all application processes, to preserve scalability. Coordination between processes is the responsibility of the application, which may use any means

available to achieve this. Any information that DAOS needs to propagate between application processes is passed by upper layers rather within DAOS itself to maximize efficiency. For example, on open, the container handle could be shared using an MPI collective - potentially one that the application performs in any case at this time.

All DAOS operations are asynchronous and use an event mechanism similar to that used in the Portals communication library to signal completion. This enables upper levels to initiate many concurrent operations from a single thread. Unlike Portals, the event mechanism permits multiple *child* events to be aggregated into a single *parent* event to simplify the expected case of successful completion of many concurrent operations. In the case of unsuccessful completion, the child events may all be queried individually for detailed error handling.

DAOS uses multi-version concurrency control at a byte-level granularity to allow multiple versions of a container to be read concurrently by multiple distributed applications while being updated by multiple processes within a single application. DAOS concurrency control is based on the *epoch*, which are arranged in a total order such that epochs less than or equal to the highest committed epoch (HCE) correspond to immutable container versions. The current HCE of an open container may be queried at any time and the corresponding immutable version of the container will remain readable until the container is closed or the application signals it no longer requires access.

In a producer/consumer workflow, consumer applications may also wait for a new HCE to be committed so that updates can be processed as the producer commits them. The immutability of the HCE guarantees that the consumer sees consistent data, even while the producer continues with new updates.

Epochs greater than the HCE denote atomic transactions that are guaranteed to be applied to the container in epoch order, irrespective of execution order. This allows an application not only to describe transactions in which multiple distributed processes update multiple distributed objects, but also allows it to execute many such transactions concurrently. Since the storage servers provide the ordering guarantee, all updates may be transmitted eagerly to fully utilize server and storage bandwidth.

Transaction commit is performed by the process leader to atomically make all updates in all epochs after the current HCE up to and including the commit epoch, durable and visible. By committing, the process leader guarantees that all updates in the transaction have been applied to its satisfaction (e.g. the application may have chosen to remove a container shard addressed by failing writes). If the commit is successful, the specified epoch becomes the new HCE.

Uncommitted updates, including any still in flight, are abandoned when the container is closed. This ensures that unexpected termination results in transaction abort. The application may also explicitly close the container to abort current transactions since updates using the closed container handle will fail. More details are available in [D5].

3.3 IOD

The I/O Dispatcher, IOD, has two main goals - to provide an "impedance matching" layer between the diverse abstractions and workloads presented by the layers above to the relatively simple abstractions and performance capabilities of the layer below and to manage the burst buffer.

3.3.1 IOD Burst Buffer Management

Forecasts show that disk storage can economically provide the capacity require at exascale but not the performance and that flash storage can economically provide the performance but not the capacity. Although it is well known that flash is more economically efficient for IOP's performance, it is equally true, albeit less well known, that flash is also about twice as economically efficient for streaming bandwidth (i.e. GB/s/$), as shown by economic analysis done by Gary Grider in 2009[1]. This is important since it has been shown that log-structured, I/O decoupling software like PLFS, can convert most typical HPC workloads into well-aligned independent data streams. This ability to efficiently capture updates into a relatively low capacity but high performance storage tier not only frees the application to continue computation but also allows the captured data to be redistributed and aggregated as it is streamed to high capacity, but relatively low performance disk based storage.

Although there is broad consensus about the need for a burst buffer tier, it is not yet generally agreed where flash memory should be positioned within the existing HPC storage architecture. There are three primary possible candidate positions: on the compute nodes themselves, within the storage system, or somewhere in between. In this project, the third option was chosen for several important reasons: failure domains, network provisioning, and in-transit analysis which are described below.

Placing the flash burst buffers directly on the compute nodes significantly increases complexity by intermingling the failure domains. This intermingling can be easily understood by considering the base-case of using burst buffers for checkpoint operations [BENT-MSST12]. When the burst buffers are not local to the compute nodes, an interruption in the computation can be easily recovered by restoring the last checkpoint from the independent burst buffers. Similarly, a failure in the independent burst buffers will not affect the currently running computation, which can simply store its next checkpoint into the remaining burst buffers. It is only in the presumably very rare situation in which there is concurrent interruption of both the burst buffer and the computation layer, in which the computation must perform a more expensive restore from the slower disk-storage tier. The instance of concurrent interruption however should be extremely infrequent since there will be approximately a 100:1 ratio between compute nodes and burst buffer nodes. With similar MTTF of the compute and burst buffer nodes, this suggests that only 1% of compute interruptions will be adversely affected by a concurrent burst buffer failure. The implication is that independently located burst buffers need not be failure resilient since there will only very rarely be a need to recover from burst buffer failure. Note that the above assumes a resilient fault-tolerant layer above IOD. Either the Mercury client/server connections between CNs and IONs will need to be resilient to transparently mask interruptions in the IOD layer or the application must provide this capability itself. The current implementation, which binds the IOD processes and the Mercury servers into a single MPI namespace cannot survive burst buffer failure.

Conversely, consider the scenario in which burst buffers are collocated with compute nodes. In this scenario, every compute interruption will cause a simultaneous burst buffer interruption meaning that the application must recover from two failures: a failure in the computation and a failure in the burst buffers. Therefore, non-resilient burst buffers will require that every compute interruption be recovered from the slower scratch tier. To prevent this slower recovery, collocated burst buffers would need to be resilient either via complex distributed erasure coding or simpler mirroring or other methods.

[1] http://www.hpcuserforum.com/presentations/santafe2014/Gary%20Grider%20LANL.pdf

However, new solid state storage technologies arriving with new generations of supercomputers may drive future architectures towards placing the performance storage tier on the CNs. The EFF project has therefore attempted to create flexible software that will work even with collocated burst buffers. It has not yet considered how to provide resiliency in the burst buffer layer beyond considering that a distributed, block-layer erasure coding, storage system would remove the need for IOD to be itself resilient. Related work has been done at Livermore in the Scalable Checkpoint/Restart library which has shown how to erasure code saved checkpoints from one compute node across its neighbors thereby allowing local restart.

The motivation to keep burst buffers independent from the storage arrays in the scratch storage system is simpler: one, to avoid similar intermingled failure domains, two, to allow the storage network to be provisioned for the slower disk-based storage and not the faster burst buffer, and three, to provide a separate layer for in-transit data analysis. Therefore, although IOD is flexible enough to run on compute nodes or on the storage nodes, it is primarily designed to run in the middle, at the intersection of the high-speed interconnect and storage networks. For example, deliverable 8.2 showed that analysis operations can read data directly on the Lustre object storage servers via the IOD interface.

Complex workflows, such as producer-consumer, also may be best served with independent burst buffers located between the compute nodes and the storage system. In-transit analysis on data that is already streaming from the compute nodes to the storage system allows for different design of compute nodes and analysis nodes. This may result in a more efficient workflow exploiting nodes with different physical resources such as processors optimized for these different workloads. Also, independent burst buffers prevent interference between the analysis and the computation which would be more likely if the burst buffers were located directly on the compute nodes.

Finally, the burst buffer layer was chosen to explicitly expose data movement decisions to the application. This is to ensure that the best, most intimate, application-level knowledge of the workload requirements inform the movement decisions. It is noted that, historically, similar data declaratives like those found in IBM's JCL and memory overlays were ultimately abandoned for solutions that automated data movement. Hadoop Map/Reduce executes map tasks local to the data they operate on however HDFS allows transparent data movement of non-local data. Similarly, the burst buffer API only insists that the user or application decide when and how data moves between tiers but does not insist that the individual read and write operations specify the data locality. In other words, IOD, like Map/Reduce and HDFS, will perform best when the user has positioned the data such that the compute will find it locally but will still perform correctly, albeit more slowly, when the data is remote. The IOD data movement API is shown in Figure 4 below.

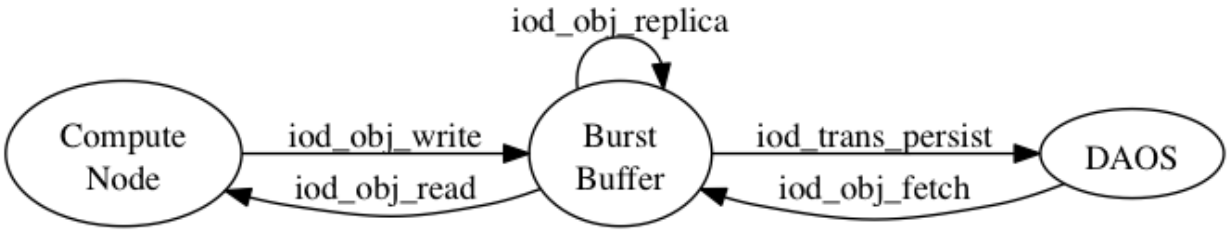

Figure 4: IOD Data Movement API

3.3.2 IOD Storage API

In addition to a burst buffer management API, IOD includes many other capabilities not present in the POSIX interface. It transitions to a simpler object storage API to reduce the namespace complexity of current file systems and allow easier distributed metadata management. It adds more complex objects as first class storage abstractions: arrays and key-value stores. Array objects allow the user to specify dimensionality descriptions of large multi-dimensional data structures and IOD will be responsible for the complexity of unraveling and rewinding these arrays to and from one-dimensional byte streams. Key-value (KV) stores provide a simple interface by which users can effectively manage huge amounts of their simpler key-based data, such as how HDF5 uses IOD KV stores for metadata and ACG uses IOD KV stores for storing graphs (through the HDF5 Map object interface).

Other key features of the IOD storage API are asynchronous operations, list I/O, and end-to-end data integrity. Asynchronous operations allow applications to pipeline their I/O with their computation but, perhaps more importantly, they allow the application to move multiple I/O operations into IOD to enable IOD to more effectively optimize scheduling and aggregate multiple requests into a smaller number of requests. Note that the current software does not yet take advantage of this ability; this API was added to allow these important future optimizations. Not only is each I/O operation asynchronous at the IOD interface, but the distributed transactions are also asynchronous: multiple processes can participate in the same transactions at different moments in physical time. Presumably, due to increased jitter, new program models will be developed to allow asynchronous parallel computation. These programming models can use distributed, asynchronous transactions to take multiple data operations that are distributed across both space and time and group them into logical atomic operations. The storage system then ensures consistency of these groups of modifications on transaction commit.

IOD allows all operations to take lists of I/O, instead of having the application issue each I/O individually. This is increasingly important in an environment in which file system clients are not collocated with the issuer of the I/O operations but are rather connected via an I/O forwarding layer. Since the file system client can buffer multiple small I/O operations, when it is collocated with the application, these multiple small I/Os will not overly adversely affect performance. However, with an I/O forwarding layer that synchronously ships every I/O operation across a network, an application that issues multiple small I/Os can adversely affect performance. IOD therefore allows such applications to describe multiple I/Os in a single operation, thereby reducing the number of network round trips.

Finally, the IOD API allows the application to pass checksums for each of its data buffers, enabling true end-to-end data integrity all the way between physically stored data and the application's own memory preventing the silent data corruption that would otherwise imperil the correctness of all exascale applications.

3.4 HDF5

HDF5 is an I/O middleware package that is widely used by many DOE application codes to store both checkpoint and analysis data. HDF5's programming API provides a set of user-level object abstractions for organizing, saving, and accessing application data in a storage container, such as groups for creating a hierarchy of objects and datasets for storing multi-dimensional data. HDF5 also allows applications to add attributes to groups and datasets to track application metadata, and to create links between groups and objects, allowing

objects to be shared between groups in the container hierarchy. Applications written to use the HDF5 API can run on a broad range of platforms, from today's petascale HPC platforms to laptops, with minimal modifications to their source code.

The HDF5 package has historically consisted of three major components: (1) the HDF5 data model objects (e.g., groups, datasets) exposed by the API, (2) the HDF5 binary file format used to store the objects in byte-streams (typically POSIX files), and (3) the HDF5 library, responsible for translating between the data-model objects and the byte-stream representation, and for implementing a variety of caching, concurrency control, and layout optimization strategies to enhance performance.

The approach for the EFF project was to build on the success of the existing HDF5 package, retaining features that have proven valuable and scalable, while addressing limitations in the areas of scalability, performance, and reliability that have been especially challenging on POSIX-compliant file systems. Together with the other partners, The HDF Group worked to design, implement, and use new transaction and object-store capabilities as part of this prototype project.

The HDF5 API was kept largely intact, in acknowledgement of its well-accepted object-model and with an eye toward providing an easy transition for existing applications (and higher-level libraries) to the EFF stack.

The HDF5 binary file format is no longer used in the EFF stack. Each HDF5 data model object is now represented as a set of IOD (KV, Array, Blob) objects, as described in [H3], and IOD KV objects are used to store HDF5 metadata, replacing binary trees that index byte streams. A modularized version of the HDF5 library that supports a Virtual Object Layer (VOL) was used in the EFF project. This version was developed prior to the EFF project with funding from LBNL, but is not yet part of the standard HDF5 distribution. For the EFF project, a specialized client/server VOL plug-in that interfaces to the IOD API was developed, replacing the traditional HDF5 storage-to-byte-stream binary format with storage-to-IOD objects.

Caching and prefetching is handled by the IOD layer, rather than by the HDF5 library, with the HDF5/IOD VOL server translating an application's directives for HDF5 objects into directives for IOD objects. Whereas HDF5 traditionally provided knobs for controlling cache size and policy, and then tried to "do the right thing" with respect to maintaining cached data, the EFF stack relies on explicit user directives, with the expectation that written data may be analyzed by another job before being evicted from the burst buffer.

In addition to the changes 'beneath' the existing HDF5 API, the HDF5 package was extended with new features seen as critical to future exascale storage needs: asynchronous operations, end-to-end integrity checking, data movement operations that enable support for I/O burst buffers, and support for transaction capabilities that improve fault tolerance of data storage and allow near real-time analysis for producer/consumer workloads.

Finally, HDF5 was expanded with new capabilities that are targeted to both current and future users: query/view/index APIs to enable and accelerate data analysis, a new map object that augments the group and dataset objects, and an analysis shipping capability that sends application-provided Python scripts to execute directly where data is stored.

Section 4.4 summarizes the HDF5 EFF features and capabilities, while [H1] and [H2] cover them in detail.

In addition to the capabilities added to HDF5, The HDF Group also developed a new high-performance RPC (function shipping) package called Mercury, in collaboration with Argonne

National Laboratory and built on their previous IOFSL package[2]. Mercury is designed to be extensible to many network architectures, and provides asynchronous operations and fast bulk data transfers. More details about Mercury are available in [H4].

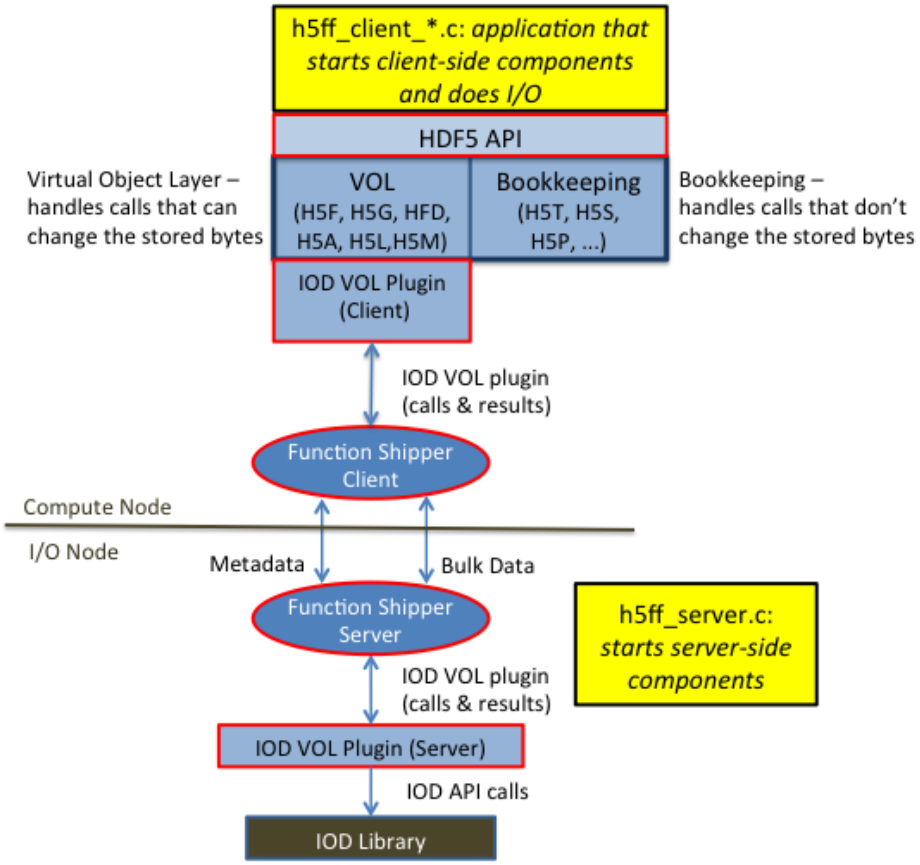

Figure 5: HDF5 EFF Components

In relation to other layers of the EFF stack, HDF5 is a user of the IOD API, and provides the entry point to the stack for the ACG application, as well as for the VPIC-IO/H5Part demonstration code and the FastBit and Alacrity indexing packages. The Mercury layer is used to ship functions and results between the CNs and IONs.

Figure 5 shows the major HDF5 components, their typical run-time location on the EFF hardware, and their relationship to the IOD library.

3.5 ACG

Arbitrarily Connected Graphs (ACG) have found application in a host of scientific and commercial computations. Over the last decade, research of the corresponding ACG algorithms created a relatively new parallel computing paradigm, namely graph-parallel

[2] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P. Sadayappan, *Scalable I/O Forwarding Framework for High-Performance Computing Systems*, IEEE International Conference on Cluster Computing (Cluster 2009), New Orleans, LA, September 2009.

computation. Graph-parallel computing is different from the Bulk Synchronous Processing (BSP) model of computation common in HPC in that it is more asynchronous in nature.

In this project, ACG applications were developed to provide a different type of workload for the EFF stack and to investigate appropriate APIs for them. An additional goal was to investigate the design of a big-data-exascale bridge allowing datasets from the growing world of big-data to be transported into a supercomputer and processed for analytics purposes. Such a bridge is interesting for two reasons. First, it can help offload the data-ingress module, a peripheral step but equally and sometimes even more compute intensive than the analytics itself, to a scaled out COTS cluster. Second, it presents quite obscure design problems from the operational differences between Map/Reduce style processing and MPI jobs on an HPC system.

A graph consists of its own basic topology information, as well as auxiliary data (network information) that might be associated with the vertices and edges. For an application, the graph-specific abstractions include vertices, edges, partitions (the collection of vertices and the edges between them), dictionary-items, probability distributions corresponding to the vertices and so on. For the EFF project, these graph abstractions are represented using HDF5 objects; for example, partitions are represented as HDF5 groups, edges are stored in HDF5 datasets, and dictionaries as either HDF5 maps or datasets.

The HDF5 Adaptation Layer (HAL), developed for the EFF project, is middleware that translates between traditional graph abstractions used by ACG-ingress and graph computation kernel applications, and the EFF stack. The HAL exposes the data inside an EFF container via queries framed in the language of graph objects and supports incremental I/O in terms of graph objects. Using the HAL APIs, an application can lay out the graph structure by pre-allocating space for partitions and other graph-objects, can add and modify objects, such as a weighted edge or a dictionary element. In addition, by providing a simple transaction-handling system within itself, the HAL APIs also shield an application from the underlying EFF paradigm of transaction-based updates, whether that is an ingress module or an analytics computation, should the application choose not to expose itself to the semantics of the transactional I/O model.

The ACG data analytics pipeline starts with raw data at ingress that must be pre-processed to extract graph structures and associated network information. These structures with graph and network information are then represented in HDF5 format on the EFF stack via calls to the HAL API, as described above. The ingress module partitions these structures appropriately so that each partition fits in the memory of the compute nodes over which processing will be distributed.

The next stage of the ACG data analytics pipeline is the execution of a graph analytics computational kernel. The analytics kernel, running on the CNs, loads the graph objects written in the ingress phase, analyzes them, and produces results that can be stored in the EFF container. The HAL again provides the interface from existing computational kernels to the EFF stack. For more details on the ACG EFF design, see the ACG design document [A2].

4 Implementation

4.1 Overview

Implementing the EFF stack prototype was a significant engineering enterprise. Although all three main layers of the stack were able to leverage existing bodies of code, all the major functionality was quite revolutionary and HPC was brand new territory for the graph analytics layer. The prototype now runs on multiple platforms and can be installed with minimal help from its developers. The following sections describe how each layer of the stack was implemented and highlights major features.

4.2 DAOS

The first implementation of the DAOS API was intended as a stepping-stone to enable development and debug of the upper stack layers as early in the project as possible. It therefore supported a limited subset of the full DAOS functionality (e.g. no recovery support), but could be used with any distributed implementation of POSIX supporting *flock()* and it was available by project quarter 3. Further development focused entirely on a full DAOS implementation leveraging Lustre. This version of DAOS was based on Lustre 2.4.0 and the completed prototype represents approximately 60K LOC of changes. Details of the prototype are provided in the following sub-sections and in [D6].

4.2.1 Storage Topology

In traditional Lustre, the metadata server (MDS) handles object allocation and assignment more or less efficiently by taking into account space imbalance between Lustre Object Storage Targets (OSTs) as well as the OST to Object Storage Server (OSS) mapping. Although explicit layout of Lustre files over OSTs is possible, the assumption is that all OSTs support similar IOPS and bandwidth and are fully resilient. This model therefore cannot support the sorts of application-driven allocation policies envisaged for the EFF stack.

In contrast, the DAOS API exports detailed information on failure domains and the locality and performance of all storage targets comprising the backend storage system through the *System Container* abstraction. A simplified description of failure domains was deemed adequate for the DAOS prototype, which represents the storage system as a hierarchy comprised from the top down in increasing order of co-dependency: cages, racks, nodes and targets. This enables upper levels of the EFF stack to make much more informed placement decisions, both when allocating new container shards and when distributing data over existing container shards.

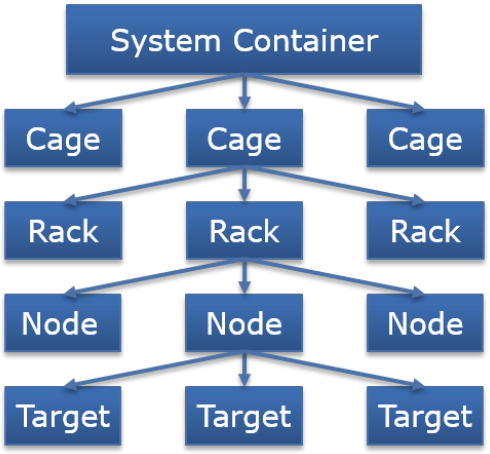

Figure 6: DAOS System Container

The system container API was implemented partly using existing Lustre APIs to return OST space statistics and network distance. The remaining OST characteristics including type, location in the system container hierarchy, bandwidth and IOPS is retrieved from the Lustre configuration metadata and a new facility was provided to allow the system administrator to store and update.

4.2.2 Container Abstraction Support

DAOS containers are represented in Lustre as a regular file visible in the traditional POSIX namespace, but with a special layout type. While POSIX operations such as `rename(2)`, `unlink(2)` and `stat(2)` work as expected, `open(2)` returns an error and DAOS containers can only be opened and created via the DAOS API.

Metadata for each container is stored on a Lustre Metadata Target (MDT) referenced by a unique identifier (FID) while container shards are stored on the OSTs, also referenced by FID. Container metadata includes regular POSIX security attributes, the container layout as a list of OST FIDs and epoch information such as the readable epochs and commit state.

Container shard addition and deactivation which alter the container layout, and epoch commit/rollback have been implemented on the MDS. The MDS is also responsible for creating capabilities which are propagated scalably to all container shards on container open and to individual OSTs on OST recovery. These are returned to the client and must be included in all RPCs to container shards to enforce security.

4.2.3 Lustre DAOS Client

The Lustre client was modified to support the new container file type. Unlike regular Lustre files where the OST objects over which data will be striped are allocated on creation, a DAOS container starts empty and is then modified to add or deactivate container shards. The Lustre client has therefore been changed to support dynamic reconfiguration without any interruption of service so that I/O can run concurrently with the reconfiguration process.

New infrastructure was also developed in the Lustre client to support asynchronous operations. This includes event queues and a progress engine using a set of kernel thread

pools. Asynchronous operations are dispatched to the thread pool, allowing the caller to return immediately. Completion is signaled by queuing the corresponding completion event and waking any user processes blocking on the event queue. Inflight operations may be aborted via their corresponding events.

DAOS bypasses most of the processing usually done in the Lustre client to manage striping for POSIX files and queues I/O operations directly to the Object Storage Client (OSC) associated with the OST hosting the relevant container shard. It is worth noting that DAOS objects are not represented directly in the Lustre client stack - the object ID is simply treated as an additional dimension of an extent address and is passed transparently through the stack along with the epoch number on all I/O operations.

Both buffered and unbuffered writes are supported. For buffered writes, each container shard OSC object caches extents in a single red-black tree indexed by the epoch number, DAOS object ID and extent start/end offsets. DAOS I/O operations neither acquire extent locks, which must be used for POSIX cache coherence, nor participate in the Lustre *grant* protocol, which is required in POSIX to guarantee buffered writes will not fail due to lack of space.

Container handle sharing, required to enable collective open in DAOS, has also been added to the Lustre client. DAOS `local2global()` exports all container information to userspace from the local DAOS kernel client, including the capability required to access the container shards. Upper levels may then share this information and import it into peer DAOS kernel clients using DAOS `global2local()` without any communication with the MDT or any OSTs hosting the container shards.

4.2.4 Versioning Object Storage Device

The Versioned Object Storage Device (VOSD) extends the features of the Lustre Object Storage Device (OSD) to support DAOS container shards and their versioning semantics. Since time was limited, the implementation leveraged the existing Lustre ZFS OSD and used ZFS's copy-on-write and snapshotting features to efficiently create immutable container versions on commit and support rollback of uncommitted updates on transaction abort. The main implementation task was therefore to support the application of writes in arbitrary epoch order rather than time order. Substantial engineering of ZFS was required to make this efficient.

A DAOS container shard is represented in the ZFS file system underlying the VOSD by the following entities:

- A dedicated ZFS dataset called the *staging* dataset where writes for the next to-be-committed epoch (i.e. HCE+1) are applied. This dataset also stores the container shard object index (OI) used to map the DAOS object ID to a ZFS file.
- A list of ZFS snapshots associated with committed epochs. This includes all epochs from the earliest epoch readable by any application, up to and including the current HCE.
- A list of Version Intent Logs (VILs). Each VIL logs writes for a single uncommitted epoch to ensure they are applied into the staging dataset in epoch order.

The VOSD only allows reads from committed epoch snapshots. These are guaranteed to be consistent through the DAOS commit protocol and read-only and therefore immutable. The MDT tracks which versions of open containers must remain readable and controls when inaccessible snapshots are discarded.

Since ZFS does not directly support versioned writes, updates (write and punch) must always be applied in strict epoch order to guarantee updates in higher epochs overwrite conflicting updates in lower epochs. Updates at HCE + 1 may therefore be applied to the staging dataset on arrival since updates in all prior epochs have already been applied. Updates at higher epoch are buffered in a VIL and applied when all prior epochs have been committed.

Epoch commit is a two-step process. First all VILs for epochs less than or equal to the commit epoch are replayed into the staging dataset in a process called *flattening.* Once complete, a snapshot of the staging dataset is created and the HCE is updated.

On epoch abort, the VOSD discards all VILs and destroys the staging dataset. The HCE snapshot is then cloned to create a new clean staging dataset.

One important feature of the VIL is zero-copy operation. When logging an incoming write, space for data that will update whole ZFS file blocks is allocated immediately and simply referenced from the VIL while partial block data is buffered in the VIL. On replay, blocks referenced from the VIL are inserted directly into the ZFS metadata tree to avoid copying.

4.2.5 DAOS Recovery

The MDT performs epoch recovery when a DAOS container is first opened for write. It first checks for consistent HCE. If one or more container shards have committed to an HCE higher than the HCE on the MDT, if these share a common new HCE and all OSTs are responsive, the MDT will make one attempt to complete the commit. The MDT will then instruct all container shards to discard uncommitted epochs. Note that container shards that are unresponsive at open cannot be accessed by DAOS clients since the capability allowing access will not have been established on the relevant OSTs. The container state is returned to the client when open completes, indicating whether the container has consistent commit state. This allows upper levels to complete recovery if necessary by disabling container shards that are unresponsive or inconsistent.

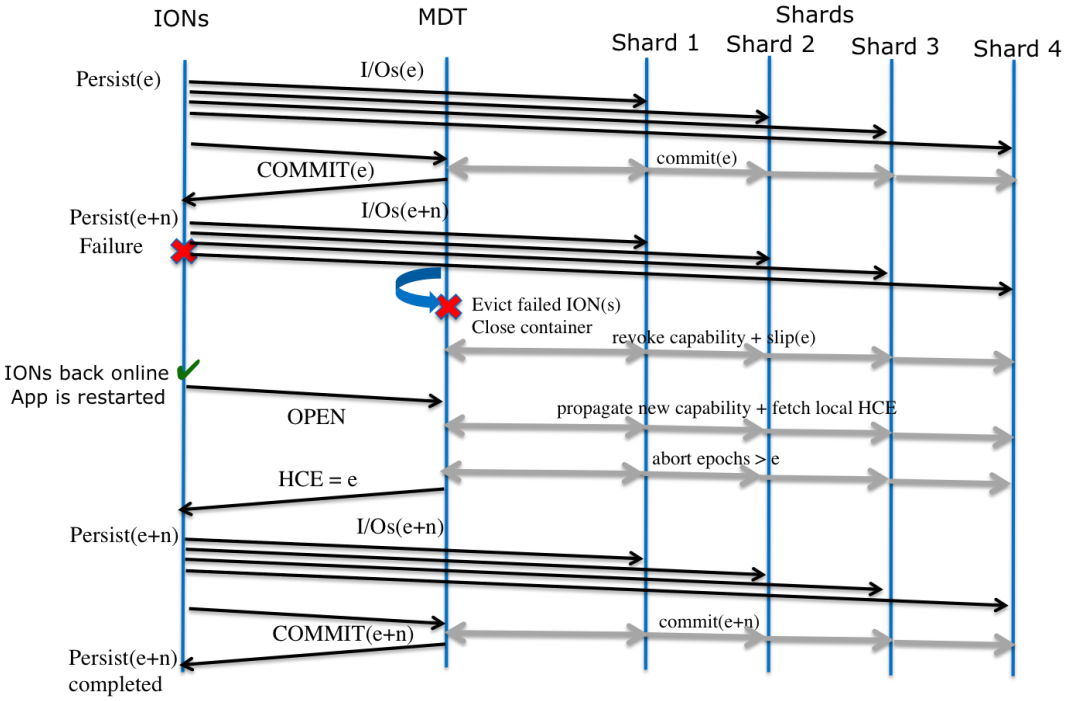

Figure 7: Recovery on Application Failure

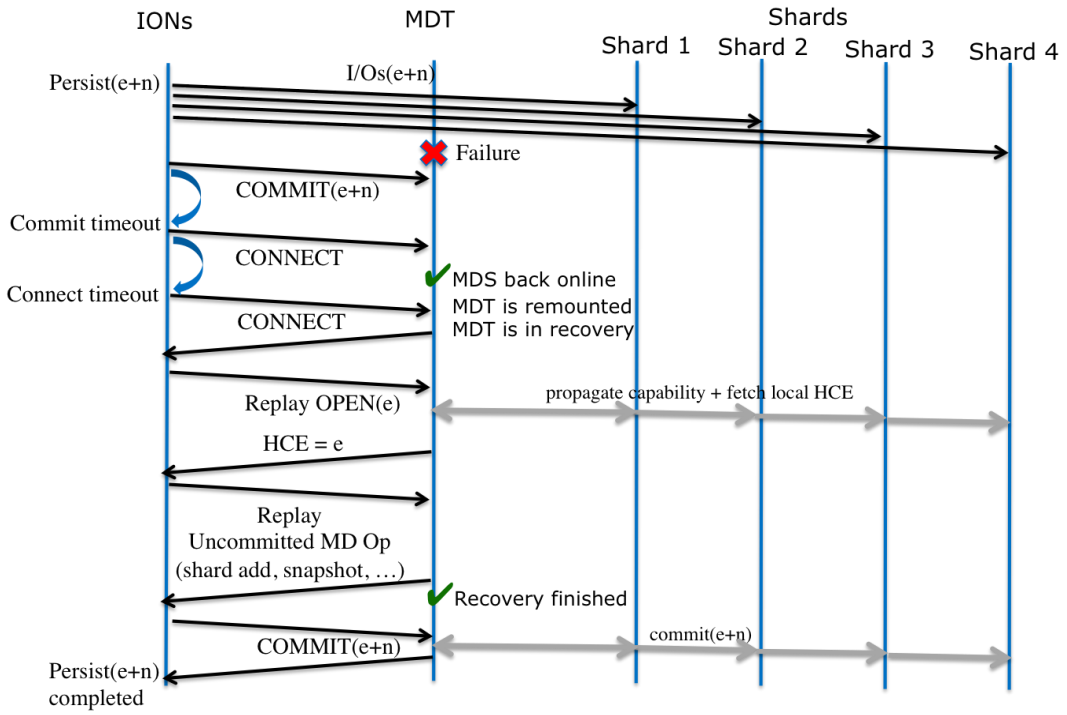

Figure 8: Recovery on MDT Failure

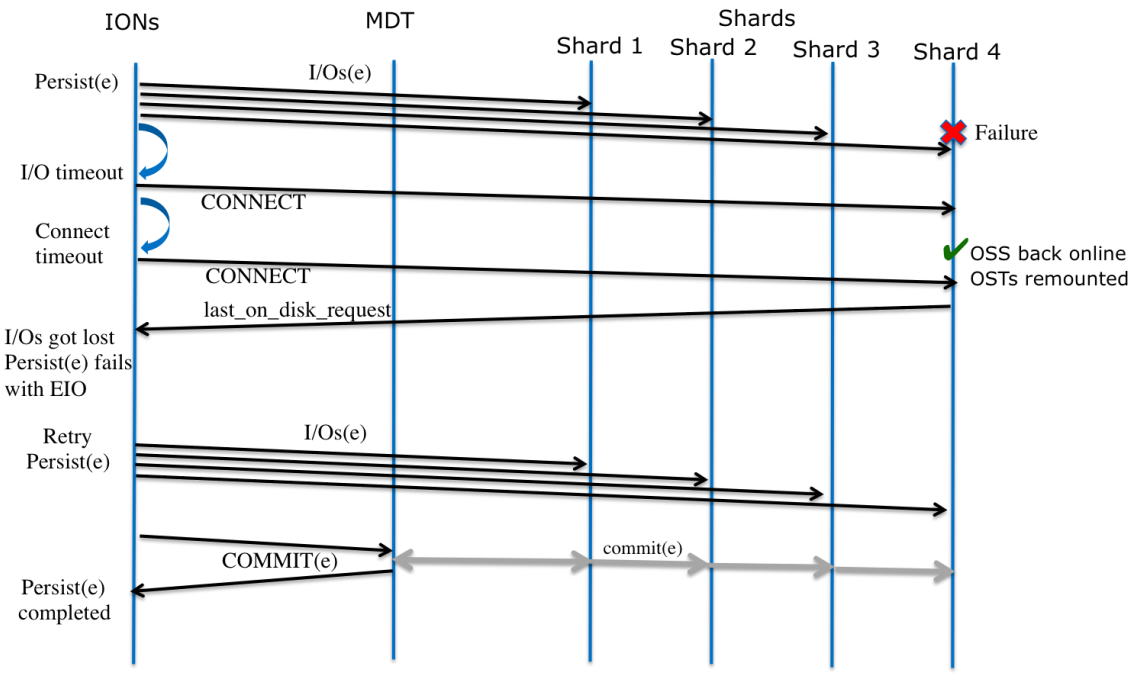

Figure 9: Recovery on OST Failure

Epoch recovery was demonstrated in Project Quarter 7 through the following test cases:

- Application failure. This may leave uncommitted updates on open container shards. These are discarded during epoch recovery as described above.
- Lustre MDS failure. These are handled via regular Lustre recovery. This restores the state of the MDS by replaying container metadata updates executed by DAOS clients but not persisted at the MDS before the crash. In-flight commits interrupted by the MDS failure are handled via epoch recovery as described above.
- Lustre OSS failure. These are not handled transparently via standard Lustre replay mechanisms as they are on the MDS. If they were, the DAOS client would have to buffer all uncommitted write data, which could quickly consume all memory or halt forward progress if it prevented write completion. However capabilities are re-imported from the MDS and connections to container shards are reestablished on clients actively accessing them. An error is returned to processes updating container shards on the failed OSS at the next opportunity and higher levels may then choose either to replay updates or abandon the entire transaction. Middleware can therefore preserve transparency to higher levels - for example IOD replays writes from the Burst Buffer.

More details about epoch recovery can be found in [D5].

4.2.6 Server Collectives

Collective communications are essential to the implementation of scalable whole-container operations including epoch recovery and commit. These require the MDT to broadcast RPCs to all OSTs hosting container shards. Simply iterating over OSTs scales O(n). Tree topologies scale O(log n), however Lustre's dependence on in-band RPC timeouts to signal failure meant that timeouts would have had to scale with tree depth. This would have proven unacceptable in large server clusters and a much more efficient and scalable method of out-of-band fault detection was required to ensure that collective communications fail promptly on the failure of any participant.

The Gossip protocol[3] is essentially a scalable, non-deterministic, and therefore fault tolerant broadcast. Participants periodically exchange shared state with one randomly selected peer to achieve global consistency in O(n) exchanges. A version of the gossip protocol was implemented in Lustre where each server's contribution to the shared state is a Lamport clock[4]. Peer health is therefore inferred within O(n) exchanges on all servers by comparing local and remote clocks. This provides prompt detection of node failure with 1000s of servers even with relatively infrequent exchanges - on the order of 10s per second to ensure minimal network load. The implementation was optimized for the expected case of relatively infrequent node failures and tested with a wide range of expected and pathological fault scenarios to ensure robustness. Please refer to [D1] for detailed design.

The Lustre Server Collectives subsystem leverages the gossip protocol to build spanning trees over healthy servers to any process on any server to broadcast an RPC to any number of services running on any number of servers and receive the combined results (see [D2] for more information). These may be performed on the fly or within established process groups to amortize setup and teardown overhead. A very flexible set of callbacks is also

[3] Patrick Eugster, Rachid Guerraoui, S. B. Handurukande, Petr Kouznetsov, Anne-Marie Kermarrec., *Lightweight probabilistic broadcast,* ACM Transactions on Computer Systems (TOCS) 21:4, Nov 2003.
[4] Leslie Lamport, *Time, clocks, and the ordering of events in a distributed system,* Communications of the ACM 21 (7): 558–565, 1978.

available to enable results to be combined both within the RPC reply and through bulk RDMA for larger results.

Two tree topologies with arbitrary branching ratios were implemented, balanced and k-nomial. The specific topology used for any process group is determined by the set of servers on which they are placed, the tree type and its branching ratio. An error is returned to the RPC caller if a server in the process groups is dead at RPC launch or if one dies during RPC execution. The caller may then either adjust group membership or retry when failing nodes have recovered.

Minor differences in performance between tree topologies were seen during testing; however they outperform direct one-to-many communications with as few as 12 servers when the network was loaded with other traffic.

4.3 IOD

It is overly pithy to describe IOD as the exascale evolved version of burst buffer mode PLFS, although it is also not entirely inaccurate. Layering above PLFS, IOD leverages much of the functionality of PLFS while adding a tremendous amount of new functionality of its own.

In addition to its own functionality, IOD has extended the PLFS burst buffer API to add all of the key exascale features of EFF: support for multiple object types, asynchronous I/O, data integrity, list I/O, and asynchronous distributed transactions. Further, IOD has addressed some of the challenges associated with overly copious amounts of PLFS metadata by using a simple striped layout for data in DAOS as opposed to the PLFS log data formatting. Readers are encouraged to consult the EMC EFF project documents and presentations for further details on the features summarized below.

This version of IOD uses modified versions of PLFS 2.3.2 and MDHIM 0.1. The completed IOD prototype has approximately 50K LOC with another 15K testing LOC. Additionally, approximately 3K and 10K LOC in PLFS and MDHIM respectively were changed as well as another 4K LOC changes in the PBL-ISAM library used within MDHIM.

4.3.1 Asynchronous and List I/O

All operations in the IOD API are asynchronous with the exception of *iod_initialize()* and *iod_finalize().* Almost all operations in the IOD API have list variants in which multiple operations can be described in a single call to IOD. The asynchrony is important both to allow the application to pipeline I/O and computation as well as to allow the storage system to accumulate a large number of operations at a time, in order to increase the potential for aggregation and other scheduling optimizations. List I/O is important for the latter reason as well as to reduce the number of network crossings since the application issuing the list I/O may not be on the same physical node as the IOD process but is rather linked via an I/O forwarder such as Mercury.

4.3.2 Semantic Objects, Migrations, and Transformations

To ease the users' interaction with storage, IOD supports *blobs* which are simple single-dimensional streams of bytes analogous to POSIX files as well as two types of semantic objects: *key-value store* and *array* objects. These provide a simpler interface for the user who wants this functionality but does not want to be responsible for converting these

complex object types into serial streams of bytes. The burst buffer management API in IOD provides rich semantic interfaces by which users can both dictate and query the semantic location of objects, both between tiers (e.g. persist this KV from BB to DAOS) as well as across multiple storage devices (e.g. create a replica of this array such that it is stored in column-order striped across this set of IONs).

4.3.3 End-to-end Data Integrity

Due to the expected tremendous volume of exascale data, existing data protection mechanisms, such as storage-based ECC, will fail at increasingly high rates thereby introducing dangerous silent data corruption. Therefore, the IOD API has checksum parameters for every data buffer operation for all three IOD data objects. The IOD implementation performs required checksum recomputation when application I/O is misaligned with respect to IOD's checksumming schema. The possible race conditions involved in these checksum recomputations have been studied and the implementation has been tested to show that they are carefully avoided. Further, the IOD implementation has created the notion of virtual DAOS container shards to allow simple and fast mappings of checksum metadata to the data that it protects.

The PLFS metadata has been augmented to include these checksums and performance tuning has shown the importance of not creating overly large checksum blocks due to the high checksum recomputation costs of small unaligned reads within the larger checksum blocks. This was a bit more complicated than merely adding a new field to the existing PLFS index entries due to the need to handle partially overwritten data. PLFS, without checksum support, will create trimmed versions of the index entries in memory when they are partially overwritten. With checksum support, PLFS could then recompute the necessary checksums for the new index entries but this is overly aggressive since the entries may not be read and the recomputation will have been done unnecessarily. Therefore, the new PLFS maintains both the trimmed index entry for fast data lookup as well as the full original entry which is used for checksum recomputation should it prove necessary as is shown in Figure 10. Also shown in Figure 11, is how IOD also implemented checksum support for KV objects stored in MDHIM. For KV objects, IOD did not modify MDHIM as it did PLFS; rather it added checksums transparently to MDHIM by merely storing the checksum for both the key and the value as a header prepended to the value.

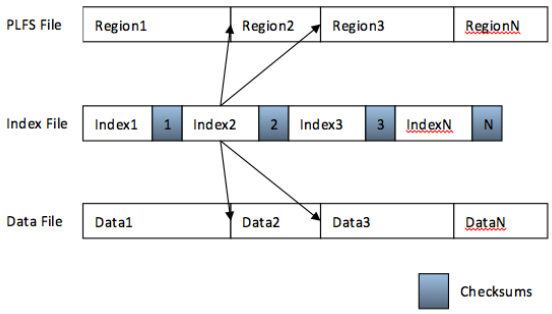

Figure 10: Adding Data Integrity to PLFS

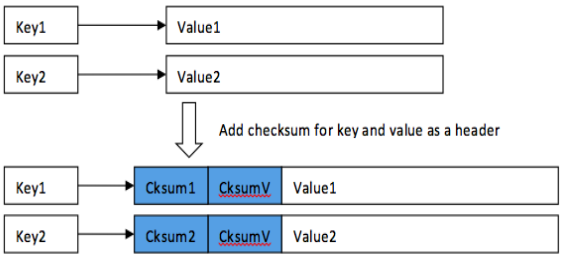

Figure 11: Adding Data Integrity to MDHIM

4.3.4 Burst buffer management

The IOD API extends the PLFS burst buffer API with all of the main EFF features. In addition to simple persist, purge and fetch operations, the IOD burst buffer API has rich, descriptive semantics similar to HDF5 hyperslabs and MPI file views to describe the placement and striping of arrays as well as semantic support for burst buffer operations on KV objects.

4.3.5 Asynchronous Distributed Transactions

The IOD implementation allows applications to manage transactions in one of two ways. In the first, the application can nominate a single transaction leader to start and finish IOD transactions for its peers. This model simplifies the IOD transaction work since IOD need not reference count the number of threads participating in each transaction. However, this model places the onus on the transaction leader to wait to finish the transaction until it knows that all of its peers have finished. In the second, IOD transactions are started and finished by all threads. In this case IOD will do the reference counting necessary to know when each transaction is finished. When operating in the latter mode, IOD requires that each thread starting a transaction must correctly inform IOD about the total number of threads that will participate, so that IOD will not inadvertently mark a transaction as finished if some slower threads have yet to start it. This asynchrony extends such that IOD even allows transactions to finish out of order but delays their readability until all earlier transactions are either aborted or themselves finished. Figure 12 below shows the IOD transaction states; some of the more complex states such as *Stale* are not described here for the sake of brevity; their full description and explanation can be found in the IOD design document.

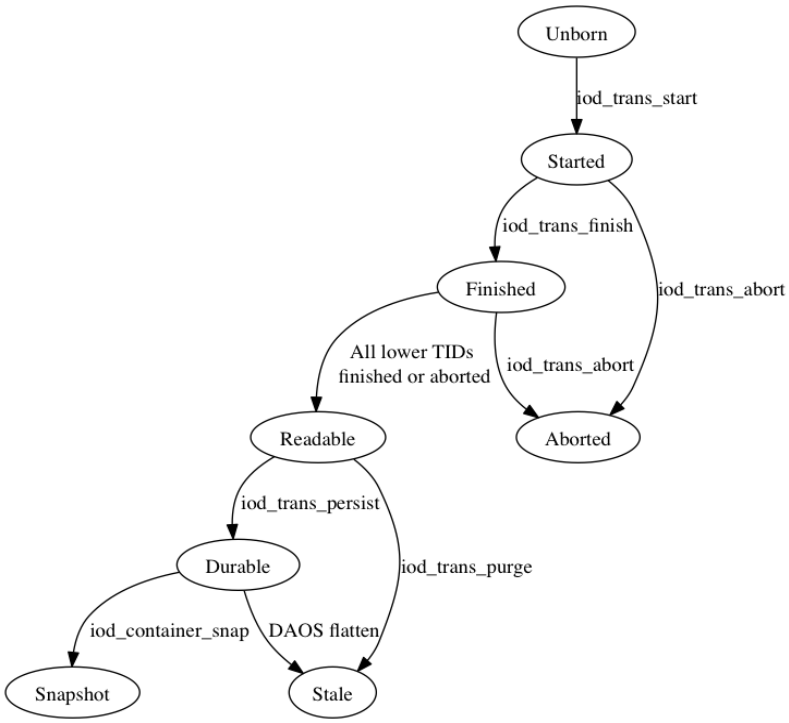

Figure 12: IOD Transaction States

4.3.6 Storing IOD Data in Burst Buffers and DAOS

Although IOD uses PLFS to store blob and array objects in the burst buffers, it does not use PLFS to store them on DAOS. For storage of KV objects, IOD uses MDHIM to store them on both burst buffers and DAOS. MDHIM, Multi-dimensional Hierarchical Indexed Middleware, is a software library that uses MPI to link multiple local key-values stores to create a global sorted key-value store.

MDHIM required modifications to add an abstract storage layer. MDHIM itself does not store and retrieve key-value data; rather MDHIM adds MPI sharding across multiple local key-value stores. The implementation of MDHIM used by IOD in the EFF project uses PBL-ISAM for its local key-value stores. Therefore, an abstract storage layer was added within the internal PBL-ISAM code used by MDHIM. This abstract storage layer then maintained support for POSIX storage to allow MDHIM via PBL-ISAM to store data into the burst buffers using POSIX I/O. To enable MDHIM to store data to DAOS required adding the DAOS API into the newly added abstract storage layer. A difficulty here was the epoch semantics of DAOS, which were unexpectedly problematic for PBL-ISAM. It opens files in read-write mode and often will perform write-read-write modifications. Reading from an uncommitted epoch in DAOS is not currently allowed so the write-read-write behavior of PBL-ISAM was failing when IOD would persist KV objects to DAOS. This was first handled by merely copying the PBL-ISAM files in their entirety to the DAOS container shards. However, this proved inefficient when persisting multiple IOD transactions because IOD, due to the opacity of the PBL-ISAM file contents, would always send the entire PBL-ISAM files even in situations in which only very little data was modified.

Clearly, IOD needed to support incremental persist in which IOD tracked which keys were modified in which transaction and would then only persist those keys instead of copying the entire tables. To do so, PBL-ISAM is run directly on the DAOS container shards using the DAOS abstract storage interface and apply MDHIM operations for the affected keys directly. This however triggers the problematic write-read-write behavior. The current implementation therefore mirrors DAOS writes from PBL-ISAM into a temporary file. When PBL-ISAM then attempts the read operations, the requested data is returned from the temporary file. One challenge remains however, which is that the requested data may not be freshly written and exist in the temporary file but is rather old data that existed before the persist operation began. Therefore, temporary files are needed to know whether particular byte ranges contain valid data or holes but this is not supported for POSIX files. To solve this, the temporary file that IOD uses is a PLFS file, since PLFS metadata knows exactly which byte ranges contain valid data and which are holes. Using these temporary PLFS files, IOD then redirects PBL-ISAM reads as follows: if the request data is found in the temporary PLFS files, return it from there, otherwise, return it from the DAOS HCE which maintains the previous version of the PBL-ISAM files.

Storing arrays is done simply by converting them to serial byte arrays and storing them into IOD blobs. One small optimization however is to adjust the default block sizes of the stored blobs to align with the array cell sizes. For example, if IOD is configured with a checksum unit size of one megabyte but the array cells are each 100,000 bytes, then IOD will use a checksum unit size for that array of 1,000,000 bytes since it is the multiple of the cell size closest to the configured checksum size of one megabyte.

To maximize parallelization of metadata and data accesses, IOD objects are split into multiple DAOS objects and distributed across multiple DAOS container shards as is shown here:

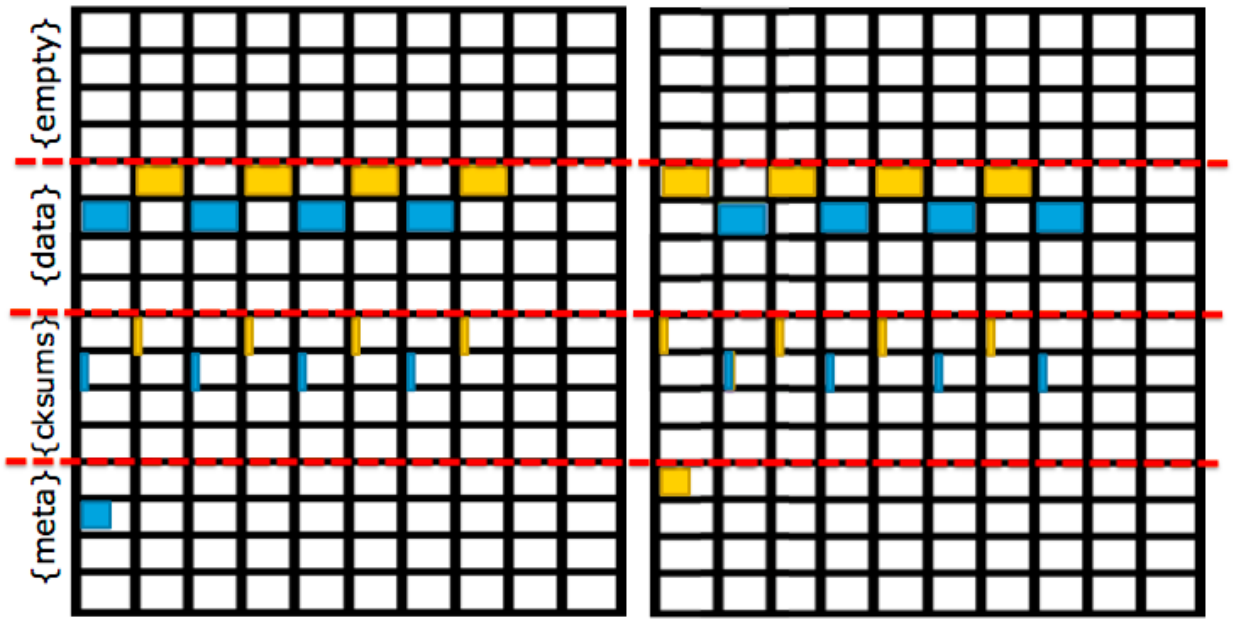

Figure 13: IOD Objects Distributed Over DAOS Container Shards

Figure 13 depicts the storage of two IOD objects striped across two DAOS container shards. The DAOS container shards are the two large grids depicting two of their three dimensions: object ID along the y-axis and object offset along the x-axis. Note that the third dimension, epochs, is not shown.

By splitting each DAOS container shard into four virtual ones, as shown by the red dashed lines, IOD creates a separate object space within each DAOS container shard for data, metadata, and checksums. Each IOD object is striped across one or more DAOS container shards depending on the total size of the IOD object. Within each DAOS container shard, IOD stores data, metadata, and checksums for each of its objects as three separate DAOS objects in each respective virtual shard. The DAOS object ID for each is found by adding a virtual shard ID of 2-bits to the 62-bit IOD object ID thereby allowing a simple mapping from IOD object identifier to the location of its data, metadata, and checksums. Note that a fourth virtual container shard is left empty although future optimizations have been designed to allow IOD to collocate its metadata and checksums into a single virtual container shard thereby reducing the number of virtual container shards needed by IOD to two and correspondingly restoring the maximum usable objects per EFF container to 2^{63} thereby enabling the large number of users needing containers to store some number of objects between 2^{62} and 2^{63}.

Two objects, one yellow and one blue, are stored as shown in Figure 13 above. Notice that the top-level metadata for each is stored is stored on a different container shard. The placement of this metadata is determined by hashing the IOD object ID modulo the number of DAOS container shards to determine which shard and then adding the two bits identifying the metadata virtual shard to find the object within that shard. For example, the blue object in this picture is IOD object number 2; therefore, its metadata is stored at offset 0 in object 2 in the metadata virtual container shard on the 0th DAOS container shard. Pseudocode for metadata location is relatively simple:

```
(dshard,dobj,doff) find_metadata(iod_obj_t oid) {
       uint64_t dshard = oid % nshards
       uint64_t doff = 0;
       uint64_t dobj = oid & IOD_METADATA_VSHARD_MASK;
       return (dshard,dobj,doff)
}
```

This top-level metadata contains basic information about the IOD object such as its last valid offset, the set of container shards across which it is striped, the size of each stripe, and the size of the checksum blocks. Arrays have additional information about their dimensionality and the size of their cells. Using this metadata, locating the data and the checksums is essentially the same as locating the top-level metadata which the exception that their masks are different and that the offset calculation is done using the stripe size and the requested offset of the IOD object.

4.4 HDF5 & Mercury

As part of the DOE Exascale FastForward (EFF) storage effort, many new capabilities were added to HDF5, some that were planned as part of the original statement of work and others that evolved over the course of the project. The main capabilities described by the statement of work focused on asynchronous operations, storage container integrity and fault tolerance, data analysis operations like query/view/index, and analysis shipping operations. As work proceeded, a high-performance RPC mechanism, a new Map object type in HDF5's data model, an asynchronous execution mechanism, and operations to manage data movement between the multiple tiers of the EFF storage hierarchy (application memory, flash on the IONs, and disk storage on the SNs) were additional identified needs. To facilitate workflow scripting environments, feature testing, and execution of analysis shipping scripts, all of the new capabilities added to HDF5 for the EFF project were wrapped in Python, extending the existing h5py package[5].

Readers are encouraged to consult the HDF5 EFF project documents and presentations [H1-H8] for further details on the features summarized below.

4.4.1 Asynchronous HDF5 Operations

Asynchronous I/O operations are designed to allow an application to schedule a read or write operation for future execution, then return immediately to the application, so it can perform other work while the I/O operation occurs. As part of the EFF project, all the HDF5 API routines that operate on containers, and thus might be impacted by a delay for I/O to complete, were extended to optionally perform asynchronously.

To enable asynchronous operations for HDF5 applications, an interface for creating 'event stacks' was added that allows multiple asynchronous operations to be bundled together for application ease of use. Each asynchronous HDF5 operation accepts an event stack as a parameter, which can later be checked for completion of all of its asynchronous events, or individual events can be extracted from the stack to check their status. The following pseudo-code outlines this functionality, showing how file open, dataset create and write operations can be executed asynchronously by an application, which then performs some computation concurrently with the I/O operations' execution and finally waits for all operations to complete:

[5] http://www.h5py.org/

```
event_stack_id = H5EScreate();
file_id = H5Fcreate("filename", …, es_id);
dataset_id = H5Dcreate(file_id, "dataset name", …, es_id);
H5Dwrite(dataset_id, <buffer>, …, es_id);
<compute>
H5ESwait(es_id);
```

An additional complexity for asynchronous HDF5 operations is the need for operations to execute in an ordered, but asynchronous manner. For example, in the code above, the container creation operation (H5Fcreate) must execute before the dataset creation (H5Dcreate), which must then execute before writing data to the dataset (H5Dwrite), but all of these operations must asynchronously execute in the proper order without further application involvement. To take this operation ordering requirement into account, the notion of *I/O dependencies* was introduced as part of the extensions to HDF5 for asynchronous operations. I/O dependencies are tracked internally within the HDF5 library, transparent to the application, and are created automatically as needed. The dependency information is sent along with the I/O operations to the ION, where the asynchronous execution engine (described below) uses them to ensure that operations are executed in the proper order.

4.4.2 End-to-End Data Integrity

In order to enable applications to have full confidence that data stored with the EFF storage stack is unmodified during the round-trip between application memory and the storage container, HDF5 was extended to provide data integrity features that validate application data over the course of its entire life. This begins with creating a checksum on data in application memory that is being added to a transaction, then transmitting that checksum with the application data to the container, where they are stored together. When the data is read back by an application, the checksum is used by IOD to verify that the data was not corrupted while stored, a new checksum is generated if the read request does not align with the stored checksum block in storage, and the checksum and data are transmitted back to the application memory, where they are once again verified, to detect errors in transmission.

In addition to the *static* end-to-end integrity process, of validating data from application memory to storage and back, HDF5 has also started to add *dynamic* end-to-end integrity steps to the data movement process. Dynamic end-to-end integrity verifies that the serialization and transformation steps necessary for storing metadata about application data (the HDF5 datatype and dataspace for datasets, data layout information, etc.) are correct and produce uncorrupted data. A detailed description of the full end-to-end data integrity process in the EFF stack was presented in quarter 5, see [EFF1].

Both static and dynamic end-to-end integrity checking are necessary to fully verify that data has been correctly stored and retrieved. But, in certain cases, an application may desire to enable or disable components of the end-to-end integrity process, for debugging or testing, or to lighten the performance burden for quick data analysis. This capability is provided to applications through the HDF5 *property list* feature, and allows application developers to control which aspects of end-to-end integrity should be applied to their I/O operations. See [H2] for all the properties to control end-to-end integrity in the EFF stack.

4.4.3 Transactional I/O

At the HDF5 level, a transaction consists of a set of updates to a container that will all become visible (readable) atomically when the transaction is successfully committed. Multiple transactions can be in progress at any given time, and transactions can be finished[6] in any order. After a transaction is finished, it will be committed when all lower-numbered transactions have been committed or aborted. Commits occur in strict numerical order. The user is responsible for managing transaction numbers, and currently the container can only be open for write by a single application at a time.

At the HDF5 level, the term *container version* (CV) refers to the state of the container after a transaction has been committed. When transaction #T is committed, the updates in the transaction are applied atomically to the container and a new container version #CV=T becomes readable. When there is an open *read context* (RC) on a given CV, the contents of that CV are guaranteed to remain readable until the RC is closed.

HDF5 uses IOD transactions to provide this functionality. When an HDF5 transaction is finished and then committed, it becomes readable from the BB. When an HDF5 container version is *persisted*, it is committed from the BB to DAOS and becomes the container's HCE at the DAOS level. The expectation is that users will commit transactions to IOD more frequently than they will persist container versions to DAOS, leveraging the BB's ability to handle bursty data and IOD's flattening and rearrangement of data before storing it on the slower DAOS disk-based tier.

When a transaction is started, the user must declare which RC to use when locating H5Objects and metadata referenced in the transaction. This is important, for example, if the user will create an H5Dataset in an existing H5Group—the metadata to locate the H5Group must be read from the container; since there are potentially multiple readable versions of the container, the library must be told what version to consult.

Figure 14 depicts the evolution of a container over time, with open RCs shown in red labeled with their respective CV numbers, and transaction lifetimes shown in green, labeled with their transaction numbers and referenced RCs. Figure 14 indicates that multiple transactions can be in progress at once, for example transactions 12, 13, and 16 between time points F and G. Multiple CVs can also be open for read at the same time; for example, after time point I, CVs 10 and 11 are guaranteed to remain readable even as updates from transactions 12 and 13 are committed to the container. Referring to transactions 12 and 13, transaction 13 is finished before 12, but since transactions must be committed in strict numerical order, the updates in transaction 13 will not be committed until 12 has been finished and committed. If the user does not explicitly indicate that they want to be able to read from CV 12, the EFF stack may optimize the application of the updates in the two transactions by "flattening" them into a single commit when transactions 13 finishes.

[6] A transaction is finished when no more updates will be added to it.

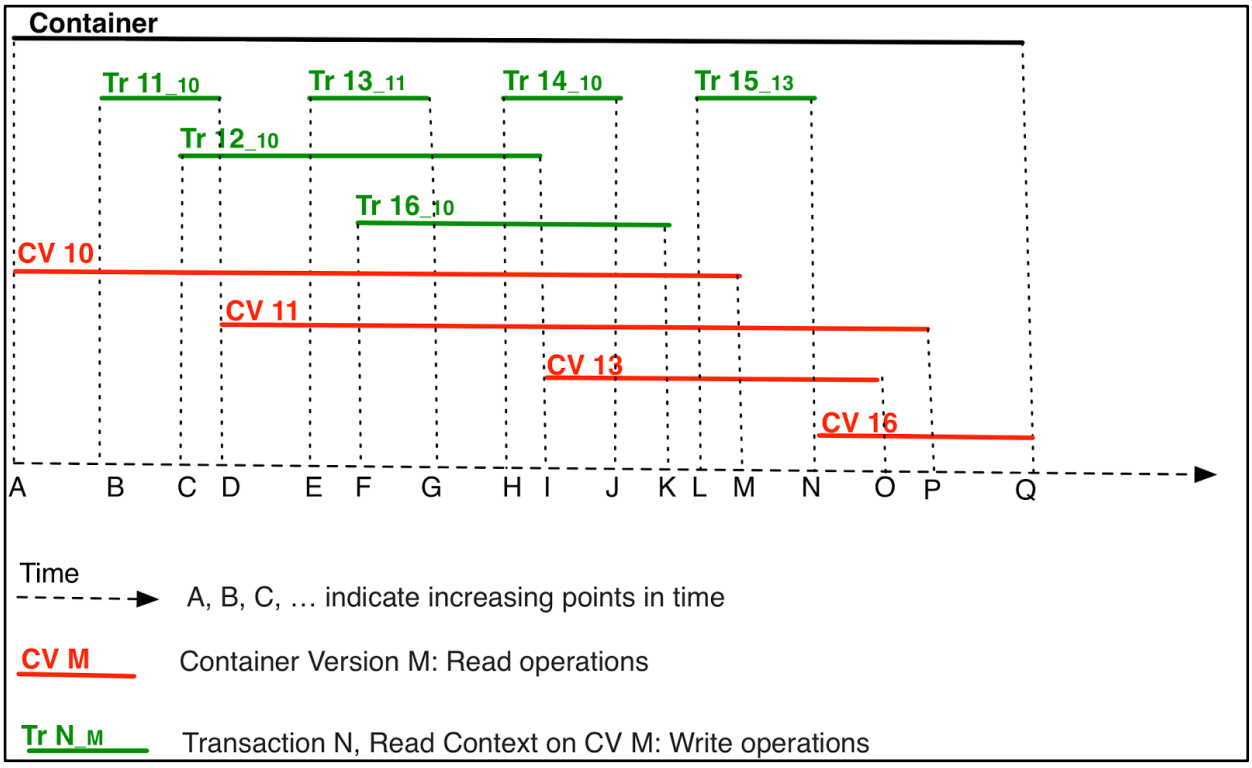

Figure 14: Container Evolution with Transactional I/O

Referring again to Figure 14, consider how the transactional model supports resilience when failures occur. If a crash occurs between time points G and H, the application can re-open the container, query to find the last committed version (CV 11), and restart knowing that the container is in a consistent state (metadata and data are in sync and not partially written). All updates made to uncommitted transactions 12, 13, and 16 prior to the failure are discarded by the EFF stack, allowing for a clean restart.

The described recovery capability was demonstrated in Quarters 7 and 8 as part of the EFF project, for recovery from CVs persisted to the DAOS tier. The design supports similar functionality for recovery from transactions committed the BB, but implementation was out of scope for this prototype phase of the project.

To support transactional I/O, new HDF5 routines were added to start, finish, abort, and set dependencies for transactions; routines were also added to acquire and release RCs, and to persist and snapshot CVs (make a named copy of the container version on DAOS).

To support transactional I/O, existing HDF5 APIs that update H5Files by creating new objects, changing contents of existing objects, or deleting objects, were augmented to take transaction IDs. For example, the H5Dwrite operation, used to update data in an H5Dataset, now requires a transaction ID. When the H5Dwrite call is issued, the data update operations for the H5Dataset are added to the indicated transaction. Those updates (traditionally thought of as writes) will not become readable until the transaction is committed. Updates to other H5Objects in the same H5File (container) made with the same transaction ID will also become readable when the transaction is committed, allowing the user to advance the state of all objects in the container in a coordinated manner. This

capability is critical for recovery from system failures, and also to provide analysis applications consistent views of all data in the container.

Existing APIs that read data from H5Files were modified to accept a read context ID. This was necessary as there are potentially multiple CVs readable at any given time. By obtaining an RC, then passing it into multiple read operations on a container, the user is able to access consistent data, even when updates to the container are still in progress.

Figure 15 shows the various states a transaction moves through, with the arcs labeled to indicate the HDF5 routine that caused the state change. The 'update' arc for a transaction in progress (or discarded) corresponds to HDF5 routines that modify objects in the container, such as H5Dwrite and H5Gcreate.

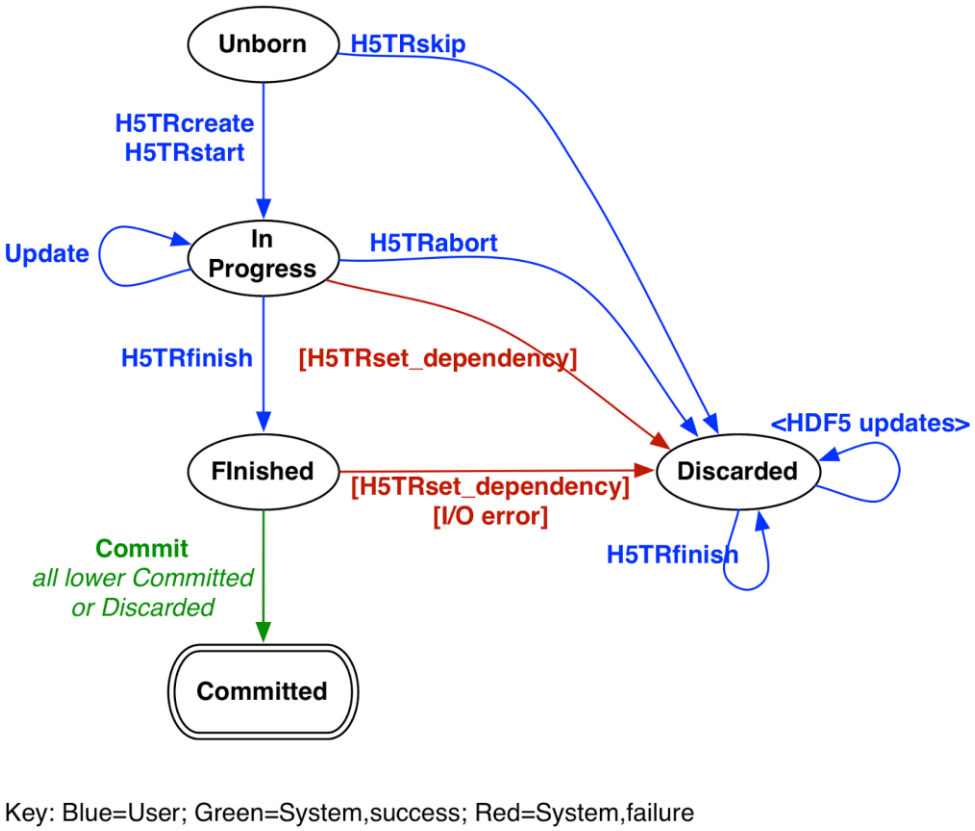

Figure 15: HDF5 Transactions States and Routines

Traditionally when an application creates an H5Object it receives an identifier that can then be used to write data to the object. With transactions, the created object doesn't exist in the container until the transaction in which it was created is committed. To support the creation and use of H5Objects in a single transaction, the IOD-VOL plugins return a *future* ID when an object is created; the future ID can be used until the transaction is committed, at which time it is transparently transformed into a normal object ID. The application rank creating the object can also use ID translator APIs similar to the DAOS `local2global/global2local` routines to share the ID with other ranks.

4.4.4 Optimizing Data Movement

The burst buffer, managed by IOD, offers many opportunities to optimize data movement from and to the application running on the CNs. As the software layer on top of IOD in the EFF stack, HDF5 acts as a translator of the user's directives to the IOD API.

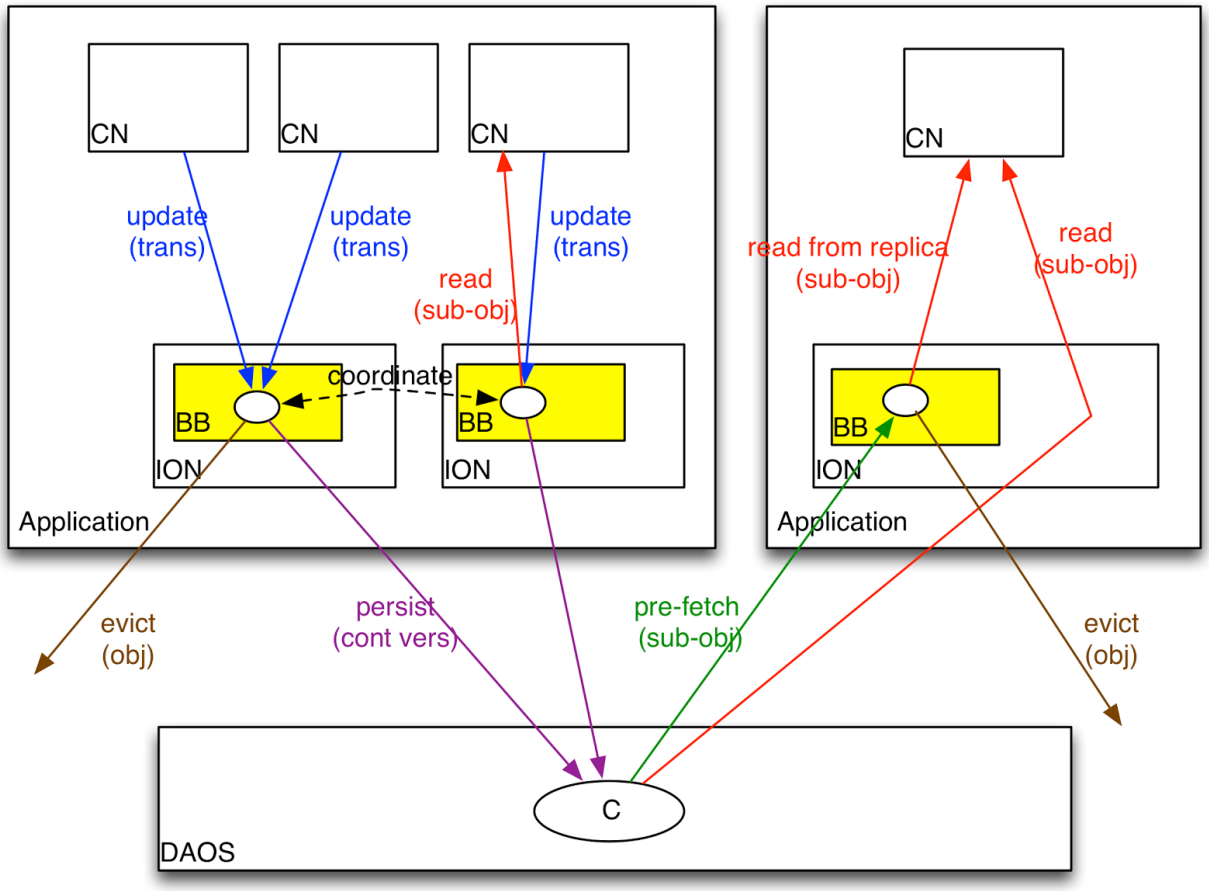

Figure 16: Data Movement Through the Burst Buffer

Figure 16 depicts the movement of data through the burst buffer and shows how the BB provides a tier where bursty data can be written quickly, reorganized, and drained off to the spinning disk of the DAOS storage layer. Both the nature of the BB (flash storage) and the IOD software (which saves changes in log format) lend themselves to supporting high-performance bursty writes.

The application controls what data is sent to the BB and when via the HDF5 API (blue arrows in Figure 16). Objects and sub-objects (e.g., traditional HDF5 hyperslab selections for H5Datasets) are the mechanism for selecting "what data" is to be updated; the HDF5 IOD VOL uses the IOD list APIs to optimize updates when irregular H5Dataset hyperslabs are specified. The application's timing of calls to HDF5 update routines (e.g., H5Gcreate_ff, H5Dwrite_ff) controls when data is sent from the application's memory on the CN to the BB. These calls can be asynchronous, allowing computation to proceed while the data transfer is in progress, and are expected to be bursty in nature. The application-initiated

transaction finish (and implicit commit when all lower-numbered transactions are finished) controls when the updates in a given transaction become readable from the BB.

The persist operation (H5RCpersist), which can also be done asynchronously, initiates the transfer of the data for a given CV from the BB to the DAOS storage tier. As part of this transfer process, IOD coalesces the individual updates for a given CV, as well as updates across multiple CVs when each is not persisted individually, and optimizes the write patterns to the DAOS storage tier. This happens transparently to the user, although the design allows for the application to supply layout hints when an HDF5 object is created that will guide IOD's placement of the object data on DAOS.

On the read side, the BB provides another level in the cache hierarchy of the system, where data in recently committed CVs can be accessed and analyzed, where data on disk can be prefetched in anticipation of use by the application, and as an extension of CN memory for out-of-core computations within the transactional update model.

The application controls what data to read and when via the HDF5 API (red arrows in Figure 16). The specification of an H5Object and sub-object (e.g. traditional HDF5 hyperslab selections for H5Dataset) in combination with an RC provides the mechanism for selecting "what data at what container version". The application's timing of the HDF5 read routines (e.g., H5Dread_ff, H5Aread_ff), which may be issued asynchronously, control when the data is read. When a read is issued and the requested data is resident in the BB, as depicted by the left red arrow in Figure 16, the data is returned to the application from the BB copy. If the data is not resident in the BB, the data moves from DAOS through the ION's memory and to the application in the CN without being loaded into the BB, as depicted by the right red arrow in Figure 16.

The application can also pre-fetch data from DAOS into the BB in anticipation of future and/or repeated reads, indicated by the green arrow in Figure 16. For the prototype phase of the project, the H5*prefetch_ff APIs were implemented to perform the prefetch operation, loading the primary IOD object associated with the specified HDF5 object into the BB. The HDF5 prefetch routines return a replica_id, which can be passed as a property in subsequent H5read_ff calls to retrieve the data from the replica of the object in the BB, as depicted by the middle red arrow in Figure 16. The functionality delivered in the prototype is the first step toward full-featured prefetch and cache support.

Related to data movement is the need to evict objects from the burst buffer when they are no longer being accessed, so that BB space can be released and reused. In the prototype version of the stack, the user is responsible for explicitly managing BB space, with IOD preventing eviction of objects from the BB that are still needed to satisfy reads for open RCs.

The HDF5 API provides H5Aevict_ff, H5Devict_ff, … routines to evict objects. These routines, which can be executed asynchronously, take object ID, CV, and optional replica ID parameters, controlling what will be evicted. The HDF5 IOD VOL evicts all IOD objects associated with the specified HDF5 object when processing the H5evict_ff request. The prototype supports the relatively fine-grained evict capability at the HDF5 object / CV level, providing the opportunity to gain experience with patterns of eviction and ability to retain some objects in the BB for ongoing analysis without forcing all objects for a given CV to remain resident. For production, support for larger-granularity (multiple objects in a given CV, a single object at all CVs, and an HDF5-hierarchy of objects) eviction is needed.

4.4.5 Map Objects

The HDF5 Map object (H5Map) is essentially a key-value store, mapping one set of elements (the keys) to another set of elements (the values). H5Map objects are built on the IOD KV store, and offer the user a fundamentally new way of storing and retrieving their data in HDF5. H5Attributes have historically provided a name/value storage mechanism in HDF5, but attributes must be associated with other H5 objects, and their primary purpose is to store user-level metadata for the container, group, or dataset they are associated with.

H5Map objects were added to support the ACG use case, but they should offer benefits to many other applications for data that does not naturally fit into the multi-dimensional array model of the H5Dataset object. When an H5Map object is created, the user specifies the HDF5 datatypes for the key and for the value. The design supports the full suite of datatypes, but in the prototype the key must be fixed length.

H5Map objects should provide useful representations for sparse arrays, for traditional dictionary lookups, and for graph analysis adjacency lists. They may also play an important role in the transactional EFF environment allowing "appended" entries to have a known (user-assigned) lookup handle that can be doled out per-rank and inserted as a lookup-key in other objects during the same transaction.

4.4.6 Query / View / Index Capabilities

A critical component of scientific computing is the analysis and visualization of data produced from simulation applications. Although HDF5 is used by many applications to store data for later analysis, the interface used to access HDF5 data has always been oriented toward the data production side of the producer-consumer equation. As part of the EFF project, HDF5 was extended with query, view and indexing interfaces and capabilities that allow analysis applications to easily locate and extract data of interest in HDF5 containers. Providing these capabilities in HDF5 will enable more powerful analysis operations to be performed in much less time.

The query and view facilities added to HDF5 enable an application to programmatically formulate a query such as "find all the regions where the element value is between 2.3 and 4.5 in datasets named 'Pressure'", and create a view object that contains the result of executing that query against a container. The results of executing the query can be retrieved from the view object by an analysis application and used to retrieve data from the HDF5 container for analysis or visualization or for further refinement.

In addition to the new query and view capabilities for HDF5, the internal architecture of HDF5 was extended to provide an interface for adding index plugins to HDF5. Index plugins create indices for data in HDF5 containers and accelerate query execution when creating views. The index plugin interface allows third-party data indexing packages to have direct access to HDF5 data elements, as they are being written to a dataset. Index plugins are designed to be transparent to applications, and operate as a native component of the internal HDF5 data flow, in much the same way as the compression and datatype conversion features in pre-EFF HDF5. To demonstrate the index plugin interface, plugins were written for FastBit[7] and Alacrity[8], two of the most popular science data indexing

[7] *FastBit: An Efficient Compressed Bitmap Index Technology, https://sdm.lbl.gov/fastbit/*

packages. More details on the query, view, and indexing capabilities can be found in [H1] and [H2].

4.4.7 Mercury

To avoid mounting the parallel file system on every compute node, I/O operations from an application must be forwarded to the storage system for execution. In the EFF storage stack, this I/O forwarding operation is performed by a new package called Mercury.

Mercury is derived from the IOFSL package and has been developed in collaboration with staff at Argonne National Laboratory. It is designed with a flexible client/server architecture that allows a CN to forward I/O calls to any ION. Each I/O call executes asynchronously on the CN by immediately forwarding metadata about the I/O operation to the Mercury server running on the ION, and then allowing the server to access data on the CN using remote direct memory access (RDMA) calls. This approach permits an I/O call to be initiated on the CN and executed on the ION without impacting the ability of the application to return to compute intensive operations while the I/O operation completes in the background, all without using additional threads on the CN.

Mercury is also designed with extensibility in mind: both the upper and lower layers can be extended without changes to the main Mercury package. The functions that are wrapped and forwarded by the upper layer of Mercury can be extended through Mercury's use of generic function wrapping macros, allowing any routine to be forwarded easily from a Mercury client to the server. Leveraging this feature, the EFF storage stack can use Mercury to forward both HDF5 VOL operations and POSIX I/O operations from CNs to the IONs, offloading both high- and low-level I/O operations from the application to execute near the I/O-rich resources on the ION[9].

Mercury's lowest level, the network abstraction layer, can be extended to support additional network transport mechanisms without changes to the main Mercury package as well. The network abstraction layer is designed as a driver framework, which allows a small set of function callbacks to be implemented as a wrapper around network transports, which are then called from the main Mercury package on both the client and server. As a demonstration of the utility of this approach, drivers for MPI and TCP/IP have been delivered with the EFF storage stack.

Mercury provides an additional benefit through its use of RDMA transfers; it can transparently provide scatter-gather operations from the CN to the ION, using the network transfer operation as the mechanism for moving data from one representation to the other. This allows HDF5 to avoid implementing scatter-gather operations internally, speeding up I/O for the application.

See [H7] for more details on Mercury's design and implementation.

---

[8] *ALACRITY: Analytics-Driven Lossless Data Compression for Rapid In-Situ Indexing, Storing, and Querying, http://link.springer.com/chapter/10.1007%2F978-3-642-41221-9_4*

[9] Although shipping both HDF5 VOL and POSIX I/O operations simultaneously with a unified Mercury client/server combination was out of scope for the project, it is certainly possible, and both aspects have been fully implemented in standalone client/server combinations.

4.4.8 Asynchronous eXecution Engine (AXE)

As effort on the EFF storage stack progressed, it became obvious that a flexible mechanism for executing I/O operations asynchronously in the Mercury servers on the IONs was needed. As a result, the Asynchronous eXecution Engine (AXE) package was developed. AXE is fundamentally a wrapper around a thread-pool that executes queued tasks, with three principal goals: simplicity, performance, and a task dependency model.

AXE tasks are designed to be simple for developers to understand and use. Instead of creating a customized thread-pool for each asynchronous execution environment, as is traditionally done, AXE allows asynchronous tasks to be queued with just a simple function pointer and context buffer for each task. When an AXE task is scheduled for execution, the function pointer is invoked and the context buffer is passed to it, allowing an application to create straightforward state machines that are not explicitly tied together.

Performance of thread-pool engines must be carefully attended to, so that the bulk of the time is spent in task execution, not thread management. AXE is designed to minimize thread overhead by using the Open Portable Atomics (OpenPA) package[10], which uses lightweight CPU-native atomic operations instead of the heavier semaphores and mutexes implemented in pthreads. Using OpenPA allows AXE to build customized locking mechanisms that are tailored to its needs, lowering the cost of thread management down to only ~1ms per task.

Finally, the EFF storage stack needs to queue tasks for asynchronous execution, but with the requirement that some tasks are prerequisites for others. This execution order dependency could be implemented within the tasks themselves, but that would reduce flexibility and increase the likelihood of creating bugs, so the AXE package has been designed with task execution dependencies from its inception. An AXE task can be queued without dependencies, or can depend on one or more tasks completing before it is executed. This allows task dependency graphs to be easily created by applications using AXE, and executed fully asynchronously without application management.

See [H8] for more details on AXE's design.

4.4.9 Analysis Shipping

Building on the query/view extensions to HDF5 and the Mercury function shipping capability, the EFF storage stack also implements the capability to ship application-defined analysis operations to locations in the storage system for local execution on data in HDF5 containers. Analysis shipping allows an application to define query operations for data of interest (DOI) in a container along with Python scripts to operate on the queried data as it is processed, in a manner similar to Map/Reduce algorithms, with the final results sent back to the application that initiated the analysis operation.

The analysis shipping capability leverages the distributed nature of the EFF storage stack by executing the application's query operation on a single ION to determine the DOI and the storage location of each piece of the DOI, sending the application-defined Python scripts to the IONs or SNs where the DOI pieces are locally available, and executing the Python scripts on the DOI local to that node. Transferring the operations for execution on the data

[10] https://trac.mcs.anl.gov/projects/openpa/

to the nodes where the data is stored allows analysis operations to proceed in parallel locally on each node, eliminating the transfer of unprocessed data across the network.

See [H1] and [H2] for details of analysis shipping design and application usage.

4.4.10 Integration with Python

As an example of the application of the changes to HDF5 for this project, its API was integrated with Python, a high-level programming language that is gaining popularity for data analysis in science, engineering, and finance. The open-source h5py[11] module was chosen as the basis for this work since it provides a one-to-one interface to the original HDF5 API via Cython[12]. On top of the low-level interface to the HDF5 C API routines, the h5py module implements a high-level pure-Python interface that allows users to develop data analysis applications with less code or perform such tasks in a REPL[13] environment.

All the HDF5 API functions developed for this project were successfully integrated into the h5py low-level interface. The high-level Python interface was either modified to accommodate HDF5 API changes or completely developed in the case of extensions not present in the standard HDF5 library: event stacks, transactions, read contexts, maps, indexes, queries, and views.

Finally, in order to verify both the C and Python interfaces, a regression testing suite was developed. It was implemented as a typical Python unit testing framework, to be executed after building the module. All the major aspects of the low- and high-level interfaces were successfully tested.

4.5 ACG

The ACG layer has two primary components, namely graph generation, which acts like a producer, and graph computation, which acts like a consumer. They both share the same representation semantics, using a single EFF container to represent an entire graph.

4.5.1 Graph Representation

The first objective in designing the ACG layer was to determine the best graph representation that could be shared by both producer and consumer. Graphs extracted from real-world datasets have two components: the graph topology, i.e. the inter-relationship between entities, and the network information. For additional details, see ACG design document [A2].

The second objective was to keep the representation flexible enough so that the same datasets could be processed at different stages by completely different process groups, possibly over different hardware with different load balancing requirements. This flexibility is achieved by completely separating the topology information from the additional information.

The topology information is more intricately associated with the graph-parallel computational framework in the sense that there is a compute thread working for every

[11] http://www.h5py.org
[12] http://cython.org
[13] Read-Eval-Print Loop (REPL)

vertex and communicating exactly on the edges. For network-information there is no direct mapping with any compute nodes. Such a division of roles maps very naturally to the HDF5 data model via HDF5 Groups.

There are three primary groups. The topology group stores the graph connectivity. The mutables group stores associated data that changes over time and the immutables group holds data that is only consumed and never modified. The most important and most used group is the topology group which is split into multiple subgroups, called partitions. Although designing new partitioning algorithms is beyond the scope of this project, the graph representation is flexible to accommodate repartitioning at any stage.

4.5.2 The HDF5-Adaptation Layer

All interaction with a graph happens through the HDF5 adaptation layer (HAL) that provides APIs to upper levels for creating and accessing graphs represented using the underlying HDF5 data model. With these APIs, new graphs can be created, new partitions can be created, and so on. The HAL is currently implemented as a C++ library, and has been tested to work with MPI applications written in C/C++, as well as from Hadoop Map/Reduce jobs.

The HAL was initially built and tested over HDF5 files on POSIX file systems. This enabled the EFF stack to be abstracted out of the visibility of the graph generation routines and the graph computation applications. In the final phase of the project, as the EFF stack became ready for the applications to use, the switch was made to EFF-containers. Since the data model and the access APIs remained the same, the applications needed very little modification. The HAL abstracts out the transactional I/O model, and lets an application continue with its pre-EFF style graph I/O – while translating them into EFF's transactional model.

4.5.3 Graph Analytics Pipeline

In the real world, datasets never come in the form of a graph and a sequence of Extract, Transform and Load (ETL) steps are necessary to find relationships that are hidden inside. A typical graph analytics pipeline starts with raw a dataset that is subsequently ETL-ed into a graph for analysis by an analytics application. Hadoop was chosen as the primary framework for graph generation. A combination of real world datasets for building real analytics programs as well as synthetic datasets for stress-testing were used.

Figure 17 below illustrates this process starting from a real-world raw dataset, the generation of the underlying graph and the representation of the dataset.

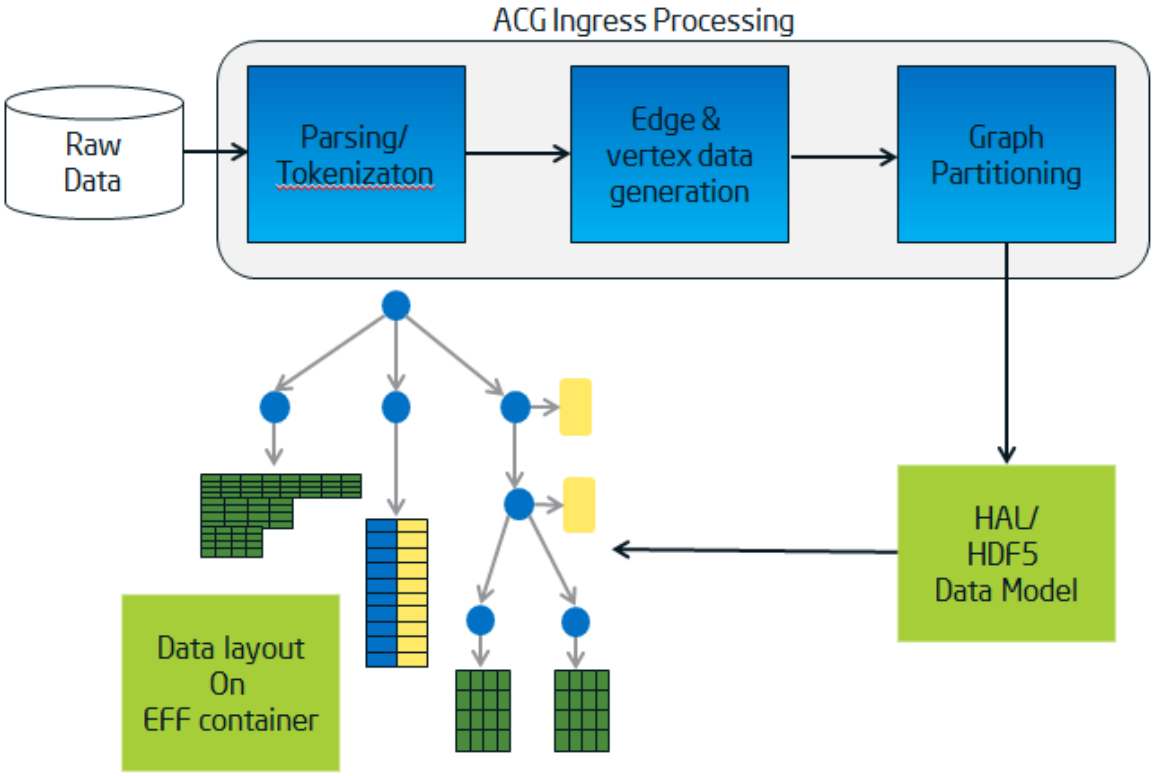

Figure 17: Graph Generation Workflow and a Graph Representation

Figure 17 also illustrates that graphs are stored in partitions which are used to load-balance the next graph analytics stage. Three different real world datasets were used to extract graphs:

- Raw Wikipedia Text. Links between pages were extracted for subsequent analysis of importance of pages through pagerank.
- Medline Data. This is a manually curated collection of abstracts of **all** research publications in the National Library of Medicine.
- MRI images. These were brain scans taken from a small set of patients

A toolset was implemented to generate Synthetic Graphs. The synthetic generator is based on the Stochastic Kronecker Graph algorithm and a faster but slightly more error-prone version called Rmat. For details, see the ACG design document [A2]. This served the purpose of stress-testing all parts of the data analytics pipeline and was used specifically to stress test earlier versions of the architectural elements that worked on POSIX. Unfortunately, time pressure did not allow specific stress testing with the EFF stack.

4.5.4 Graph Computation

GraphLab (GL) is used as the Graph computational engine. Three different applications were implemented in the GL kernel:

- Pagerank analysis
- De-noising for MRI images
- Topic Modeling using Latent Dirichlet Allocation Model over textual datasets.

For each application, the graphs are partitioned and represented in the EFF container. Each process running on CNs works on its own set of partitions. These partitions are represented in HDF5 groups and datasets. Details in the design document [A2].

4.5.5 Transaction Handling

Since big-data application-programmers are unfamiliar with transactional semantics, one of the design goals for the HAL was to keep the applications shielded from those semantics, should they choose not to use transactions. To this end, the HAL was designed to encapsulate the transactional I/O aspect for the application by implementing a simple scheme illustrated in Figure 18 below.

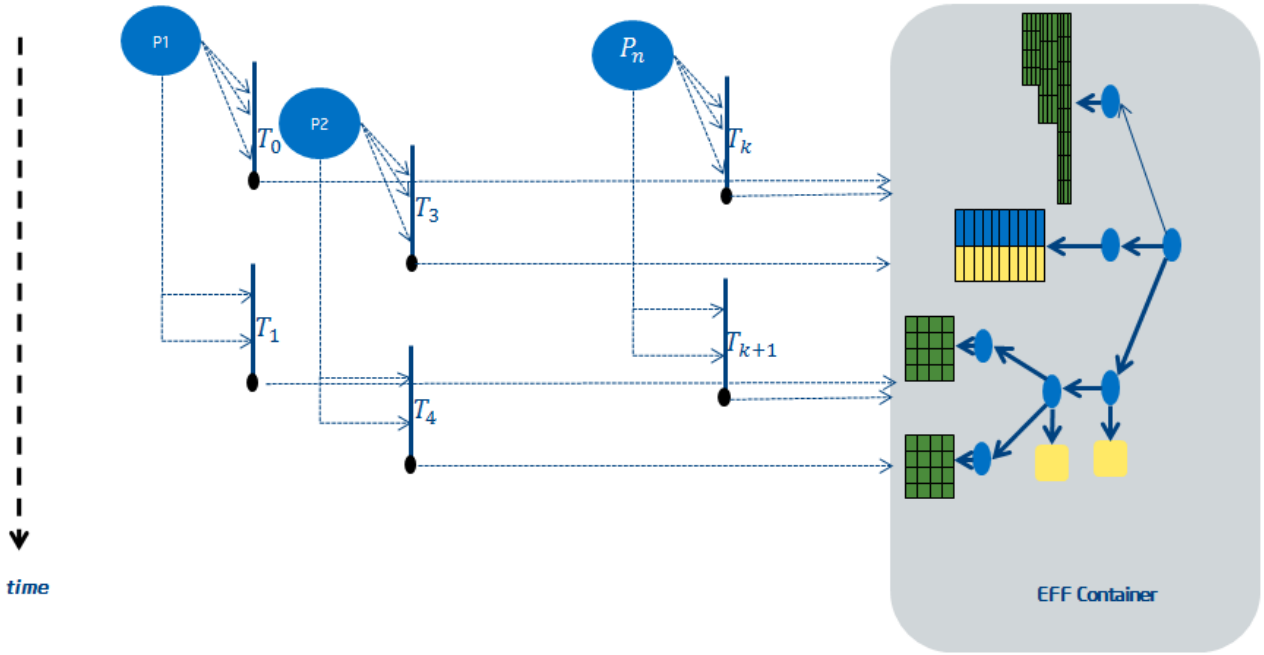

Figure 18: Transaction Handling inside the HAL

Each process has its own HAL instance mediating all interaction between the process and the EFF container. This per-process copy of HAL generates its own set of transaction numbers that obey the following rules:

1. The set of transaction numbers used by a process is uniquely determined by its rank.
2. The set of transaction numbers used by any two processes are mutually exclusive.

For example, consider a scenario with only two processes - P0 and P1, and that P0 uses even transaction numbers 0, 2, 4 … while P1 uses odd transaction numbers 1, 3, 5 … ; such a division of transaction numbers fulfills the conditions stated above.

Figure 18 illustrates the HAL operations with respect to transactions as just described. Each process sends their updates to a set of transactions that are committed to the container in pre-determined intervals. Note that no two processes bundle their updates in the same transaction. The current version of the HAL, just described, supports only a subset of what is possible in the EFF transaction model. In the EFF model, any number of processes can perform updated within a given transaction.

The HAL, being a middleware, has no knowledge of an application's intent *a priori,* and therefore restricting the transaction numbers was a natural choice. Clearly, such transaction-partitioning imposes a restriction in the ordering of commits, with potential performance implications. For example, if a lower rank process becomes a straggler, the updates from the upper rank processes do not become visible until the straggler's transactions commit. Transaction-aware applications are free to disable all transaction handling by the HAL middleware, avoiding the potential slowdowns.

5 Technical Findings

5.1 Overview

Over the course of the two year project, a prototype multi-layer I/O stack was designed, implemented, and used on multiple platforms. All participants not only had to implement their layer, but also support their users (i.e. the layer above them) or pre-existing software (e.g. GraphLab for ACG). The project experienced numerous "you shouldn't do that" comments from developers when their software was used, and "it shouldn't do that" comments from users about software that didn't behave as expected. Fundamentally there is a tension between users who want to preserve their existing methods and system developers who recognize that performance improvements are possible via changes in the programming model. The tightly integrated team improved the quality of the design and implementation, and resulted in many lessons that would not have been obvious if the layers were not used by people outside the team that developed them. These findings are presented below, with the overarching lessons learned presented first, followed by specific issues relevant to the individual layers of the stack.

5.1.1 Transaction model

The development of a massively scalable and resilient I/O model was one of the primary goals of the project. Transactions that support atomic updates to persistent data were a clear requirement in order to guarantee data consistency. However, the elimination of

unnecessary synchronization and flow control on the path to storage was also a requirement to ensure performance would not be lost to Amdahl's law.

These conflicting requirements were substantially resolved by adopting the rule that uncommitted updates should not be readable. This forces a keen awareness of such dependencies on all components of the I/O stack and applications using it, since they now must communicate updates directly when necessary, rather than via the storage system. This was not seen as a major problem for HPC applications since direct communications via an HPC fabric should have far higher performance in any case. However, it became clear during the project that programmers used to conventional I/O models may find the transactional model unnatural and some out-of-the-box thinking was required during the implementation of the middleware layers, IOD, HDF5 and the ACG libraries.

The direct consequences of making only committed updates readable are as follows:

- Applications are forced in the direction of maintaining their own cache of updates of uncommitted updates. This is simple for applications in which data structures are either immutable once created or completely updated, but more complex otherwise.
- POSIX programmers are used to local system cache making read after write operations cheap. Reuse of some conventional codes will therefore be difficult (e.g. MDHIM, which was used in IOD).

The requirement for data consistency at the application level means that the application at the very top of the I/O stack must start and finish transactions. For example, the consistency requirement for a simulation dataset encapsulating an entire time series might be that it contains only complete timesteps. Only the application "knows" what comprises a timestep therefore all levels of the EFF stack below the application must nest their consistent groups of updates within the application's transactions.

Although this allows a simple mapping from application transaction ID (and container version) all the way to DAOS epochs and reduces the amount of IOD and DAOS metadata, it means that the lowest middleware layers (i.e. IOD and DAOS) of the EFF stack cannot initiate any of their own independent transactions, and HDF5 may only initiate and complete transactions on container create and close. Furthermore, IOD and DAOS must cache their own uncommitted updates when these must be readable, which is a significant discipline. As a specific example, considerable difficulty was encountered due to PBL-ISAM attempting to read freshly written data during an IOD persist operation. To enable this, IOD currently caches its own uncommitted updates to PBL-ISAM data.

The particular form of multi-version concurrency control adopted in the project enables multiple transactions to be in flight simultaneously. This ensures that back-to-back transactions can fully utilize storage bandwidth, even in the presence of significant commit latency. However, this increases the requirement for awareness of dependencies between application processes. Furthermore, the fact that the updates performed by transactions become readable in strict order, although allowing extremely scalable implementation of the storage subsystem, means that earlier transactions can block the updates of later transactions from becoming visible, even when these transactions are entirely independent. Since the storage system is the *only* communication medium available to connect workflows and loosely coupled processes of the type found in many "Big Data" computational frameworks such as Hadoop, this restricts the types of communication that can be performed. On the other hand, the transaction model *does* fully support concurrent execution of producer-consumer workflows with very clean and consistent semantics.

Debate on allowing uncommitted data to be read and the resulting trade-off between ease of use and scalability seems worthwhile. One danger is that the current position is quite unambiguous and a relaxed position risks abuse. At a minimum, it may be worth considering allowing processes to read back their own updates on the assumption there are no conflicting updates. This could simplify POSIX code reuse if supported through client caches. Allowing processes to read uncommitted data from any peer process is more complex and difficult to scale. However, it may become tractable if the fundamental assumption that direct communication between application processes is much faster and more efficient than communication via the storage API is invalidated by new system architectures that integrate Memory Class Storage with HPC fabrics.

5.1.2 API Differences

There was much debate within the project on whether all possible snapshots of the state evolution of a container should remain accessible, effectively making the container version just another address dimension. From an HPC perspective, where time series data is common and a new timestep will most usually be saved in a single transaction, conflating version with timestep seems intuitive and space saving efficiencies will be made if these transactions do not update the whole dataset. An opposing view from a database perspective is that versions are merely a mechanism to implement transaction atomicity, consistency and durability while maximizing concurrency. The corollary is that versions that are not explicitly required should be aggregatable in the interest of performance and space efficiency.

The DAOS API takes the latter stance since the creation of ZFS snapshots is a relatively expensive operation and any opportunity to avoid it can improve performance. The DAOS transaction model therefore leans in the direction of making all transactions aggregatable by default, at the expense of assuming transaction dependence - i.e. *all* subsequent transactions must fail when a prior transaction fails. DAOS therefore guarantees only that HCE snapshots are accessible, and is free to aggregate transactions on commit and discard old snapshots when the application "slips" the epoch to a later HCE or closes the container.

The IOD API takes the former stance and the logging architecture of the underlying burst buffer storage engine, PLFS, makes retaining snapshots relatively cheap. IOD therefore makes all versions accessible and does not assume transaction dependence so that the failure of one transaction only causes other transactions that have explicitly declared dependence to fail.

The debate has provided useful insight and still continues. Should the database perspective prevail, explicit versioning via named snapshots could provide equivalent functionality, but at the expense of effectively polluting the namespace with application metadata. The assumption of transaction dependence has serious drawbacks if workflows are to be supported in which multiple collaborating applications concurrently update the same container. The different transaction semantics means that the EFF stack by default preserves all versions of transient data within the burst buffer but only preserves DAOS versions when users explicitly created named snapshots of persisted containers. This has unfortunately created confusion for developers at higher levels within the stack and made explaining transactions to application developers more difficult. Future work should weigh these conflicting requirements and determine whether the EFF stack should consolidate to a single transaction model.

5.1.3 Asynchronous APIs

All EFF stack levels provide asynchronous APIs to enable efficient overlap of compute and I/O. Unfortunately, the line of least resistance when coding, is simply to block for completion on each I/O, which defeats the purpose. To address this tendency, developers are given the ability to aggregate DAOS completion events into a *parent* event, and to aggregate asynchronous event tokens via *event stacks* at the HDF5 level. These event aggregators bundle multiple asynchronous operations into a single manageable entity. Handling failed operations within the bundle adds a layer of complexity that will be easier to take on once the I/O stack software stabilizes and the transaction model becomes more natural. Having good examples that show how to effectively use the asynchronous APIs and manage individual event failures will also go a long way toward advancing the use of this feature.

It is tempting to think that asynchronous operations can be moved high into the storage stack, with everything below that level executing synchronously. However, this should be resisted, as many levels of the stack benefit from executing operations asynchronously. This is particularly noticeable in cases where a single operation at one level starts many operations at lower levels. Executing the lower-level operations asynchronously improves resource utilization and reduces the latency of the overall asynchronous operation to the application.

For example, early in the project it was decided that IOD would call DAOS routines synchronously since IOD itself could be called asynchronously from upper layers. However, this was a mistake since the iod_trans_persist() function, even if called asynchronously, initiates a very large number of operations to DAOS that write to many objects distributed across DAOS container shards. Although the distributed IOD processes use multiple threads and aggregation to help parallelize and pipeline these operations, using the DAOS asynchronous APIs would probably be a better choice, as the DAOS aggregation is done in a kernel thread pool as opposed to the IOD user-space thread pool.

5.1.4 End-to-End Data Integrity

During performance profiling of EFF data integrity checking it was discovered that the checksum library being used was performing very poorly. Tests with different checksumming methods revealed that the method in use, *crc64*, was much slower than *adler32,* as is shown in Figure 19 below.

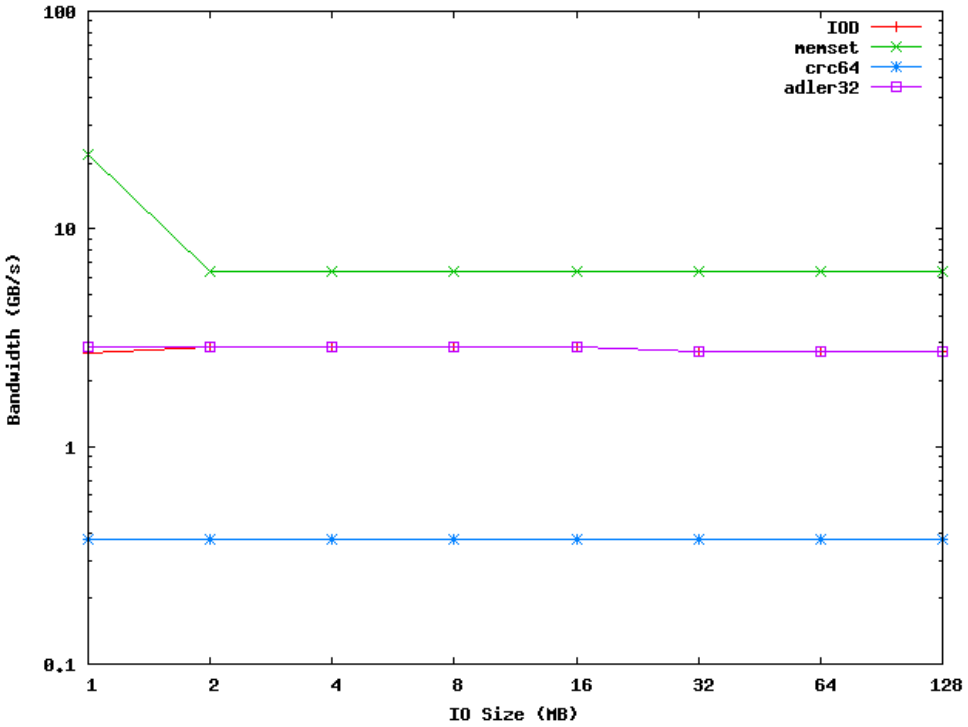

Figure 19: Performance With Different Checksum Methods

Figure 19 (note log-scale on both axes) shows that adler32 approaches memory bandwidth (as approximated by the memset routine) but crc64 is several orders of magnitude slower. The line for IOD shows that the modified EFF stack using adler32 can retain the full adler32 performance. Unfortunately, due to time constraints and the difficulty of switching between 32-bit and 64-bit checksum routines, the full EFF stack has not been upgraded to use the faster adler32 method. There was also the sense that for the data sizes anticipated, a 64-bit checksum method would be better, and hope that a high-performing 64-bit method could be identified for the production phase.

Currently both the HDF5 and the IOD APIs have optional parameters for checksums. A user application that wants data protection that it, itself, can verify, will supply the checksum to HDF5 on write and will receive it on read. Less stringent applications can choose not to provide a checksum, in which case HDF5 itself will generate a checksum, to ensure correct data transfer all the way from the memory buffer used by the application to the storage container. Note that DAOS does not currently provide a checksum parameter, therefore IOD stores its checksums as IOD metadata in a DAOS container shard. IOD is also responsible for recomputing checksums as necessary when access to data does not align with internal IOD checksum buffers. The IOD checksum recomputation and storage of checksums onto DAOS is described in more detail in Section 4.3.3.

Further exploration is needed to determine whether the current semantics are what is needed by applications. For example, adding a DAOS HA layer and putting checksums into the DAOS API will allow corruption recovery in addition to corruption detection. Also, options in the HDF5 and IOD API can be added to allow the application to determine whether data is verified during persist and fetch operations which could detect corruption earlier than waiting for the next read operation.

5.1.5 Burst Buffer Space Management

The notion to use burst buffer to absorb bursts of writes was always well-understood, having been explored in previous work by members of the IOD team. The management of burst buffer space as part of the application's cache hierarchy was less well understood, and only a fraction of its potential in this role was exercised in the prototype phase of the project.

Ambitious design discussions early in the project envisioned a number of possible methods for specifying what data to prefetch, and in some cases evict. For example, the job scheduler or application might present an intent log of the reads that would be issued, and data would be prefetched from DAOS to the BB based on that. Preliminary designs were drafted where all HDF5 objects under a given HDF5 group could be prefetched together, knowing that some applications organize data that is accessed close-in-time in this manner.

Perhaps the most powerful, and recurring, notion was that of color-coded templates, which could be applied to multiple objects (or sub-objects) in the container, indicating the sets that would be used together and should be prefetched as a whole. Templates could be created on the fly, or created, stored, and reused for multiple containers with data organized consistently. For example, there might be a low resolution "blue" template that would load multiple thumbnail images, and a high resolution "red" template that would take a location parameter and load high resolution images and measurements for data at the indicated location. Applications could easily ask for the "red" parts of the container without explicitly enumerating the objects and sub-objects to be loaded. Similarly on the evict side, evicting a set of objects that were prefetched together, or all objects in a container, or all versions of a particular object, or all objects except those that were pinned in the BB, or a variety of other set descriptors were all seen as desirable capabilities, at least by some members of the team.

While the prefetch and evict capability discussions were some of the most ambitious during the design phase, the reality of the project timeline dictated that only the basic building blocks - prefetch of the primary IOD object for an HDF5 object, and evict by HDF5 object or prefetched replica were implemented. Expressing data-movement in meaningful terms at one layer of the stack, and translating that into units-of-data-movement at a lower layer of the stack is not only a challenging 'accounting problem', but also a thorny design issue when the knowledge and control is spread across multiple levels, for example from VPICIO to H5Part to HDF5 to IOD.

The EFF stack adds powerful primitives to control data availability, but unfortunately the transactional semantics add even more complexity to what was already a challenging research question of determining what data will be needed and where and when it will be needed. The version number of the data adds another dimension to this search space, and combined with the complexities involved in tracking the lifetimes of versions, this area deserves considerable future attention.

DOE application scientists and the EFF project's ACG team are very excited by the opportunities the BB layer will provide; the mechanisms it offers to either guide or explicitly control data residency in the BB will be critical to ensure effective use of the resource.

5.1.6 Flexible Stack Configuration

The EFF stack was designed to be configured flexibly as shown in Figure 1 at the beginning of this document so that different levels of the stack can run in different locations to match compute and storage resources to the workload. Mercury provided the crucial "glue" that connected the stack across nodes and, as a modular general purpose function shipper, enabled much greater design freedom. For example, in early discussions it was assumed that the IOD API would be the correct layer to export from Burst Buffer nodes to Compute Nodes. However it became apparent that this would be quite inappropriate since the number of network round-trips could be reduced substantially by function shipping higher up the stack with an internal HDF5 API. For exascale workloads, Mercury has two important abilities. First, it has the ability to change the APIs being shipped. Second, it is fully envisaged that Mercury will allow multiple different APIs to be shipped over the same transport, benefitting future work on distributed I/O stacks, especially when a mixture of new and legacy functionality must be supported.

5.2 DAOS

5.2.1 Object Model

The DAOS object model is intentionally simple. A DAOS container appears to upper levels as a three-dimensional PGAS addressed by container shard, object and offset in which complex data structures can be embedded. However during stack integration it became apparent that a key-value store interface at the DAOS level would have greatly simplified the IOD implementation of its KV objects and internal metadata. Since DAOS aims to be a generic scalable storage API, this would be particularly true of a future IOD implementation that used also used the DAOS API to interface to Burst Buffer storage.

5.2.2 Shared Write

The original DAOS API design introduced the notion of *epoch scope.* This was intended to allow multiple concurrent non-conflicting applications to update the same DAOS container and also allow a single application to update multiple DAOS containers with the same transactional guarantees. Due to lack of time however, epoch scope was defined to be identical to a single container and only a single writing application at a time was supported in the prototype.

In the future, multiple concurrent non-conflicting writers should be supported. For example, analysis that updates the same container could be performed concurrently with a simulation that adds new data or scrubbing that checks data integrity and adds or restores redundancy at the object level. This will probably need better support for non-conflicting transactions in DAOS, including removing the assumption of transaction dependency, so that abort of one transaction aborts only truly dependent transactions.

Another class of concurrent writing application are those that don't actually change the data from the point of view of higher levels in the stack, but do change the underlying representation - e.g. to compress, or change distribution of data. Support for this capability would greatly increase the utility of the stack but require transparent cooperation between middleware instances. This cooperation could only rely on communication via the underlying stack and further research will be required to determine the appropriate programming idioms.

5.2.3 Versioning Object Storage Device

As expected, the VOSD proved to be the most critical component of the DAOS implementation. VOSD is the key enabler to multi-version concurrency control, allowing both versioned reads and I/O submissions for future epochs. Although the implementation leveraged ZFS to exploit its lightweight snapshot capability, implementing the version intent logs (VIL), required to apply updates in strict epoch order, proved challenging. Changes to handle zero copy replay turned out to be significantly complex and bug prone and in retrospect, after discussion with ZFS developers, it may have been more productive to have spent more effort on adapting the existing ZIL (ZFS Intent Log) rather than implementing the VIL from scratch. On the positive side, the VOSD prototype is now functional with zero copy VIL replay enabled, although container shard commit is still relatively expensive and further work to improve performance is required. More details about VOSD can be found in [D3].

5.2.4 Asynchronous Operations

Although the IOD prototype hasn't been able to take advantage of this capability, all DAOS operations can be asynchronous. This effectively allows a single thread to manage any number of concurrent operations. DAOS benchmarks show that a single thread per node is enough to saturate the storage system.

Currently, IOD handles persist requests via a pool of threads (actual number of threads is a tunable) reading data synchronously from the burst buffer and then writing synchronously to DAOS. This could be improved by using DAOS asynchronous I/Os, which would allow the read and write phases to run concurrently with a single thread per ION.

5.2.5 Collectives

DAOS collective open/close demonstrated excellent scalability and will definitely be useful at very large scale. Server collectives (more details can be found in [D2]) coupled with the gossip protocol (see [D1] for more information) also proved to distribute RPCs very scalably and efficiently to all container shards. It will be most desirable to take advantage of these features in traditional POSIX Lustre.

5.2.6 Arbitrary Alignment

Basing the DAOS prototype on Lustre greatly reduced its implementation effort. This however was not entirely without drawbacks. One significant problem was that Lustre I/O is very page-cache centric which means that I/O buffers and I/O offsets are page aligned and this assumption in many layers of its internal stack, including LNET, was at odds with DAOS support for arbitrary alignment.

5.3 IOD

Like the DAOS API, the IOD API remained mostly unchanged throughout the course of the project, although it did evolve a bit more than its counterpart. One example was to add an atomic append operation for blob objects that was needed by HDF5. Unfortunately, other requested API changes were not possible to implement, given the accelerated pace of the project. Perhaps the most crucial of these relate to the IOD replicas; these replicas are created either when objects from DAOS are fetched back into the BB's or when the user objects already resident in the BB's are replicated using a different layout (e.g. create a

row-order striping of this array). The current IOD API creates a tag for each replica and requires the user to use that tag in order to read from that replica. In the future, although the tag will still be required in order to purge, allowing *tagless replica* reads is preferable so that a user merely reads the object normally and IOD will transparently select which replica to retrieve the requested data from.

Another requested feature that was not identified when the project was scoped and was not possible to deliver was the ability to group subsets of objects within a container to make operations on those subsets easier. For example, many HDF5 abstractions such as H5Datasets with variably-sized cells are comprised of multiple IOD objects. A tag to allow these objects to be logically grouped for the sake of data movement decisions would have been a large simplification for HDF5.

Further, it was not fully anticipated the degree to which HDF5 would rely on IOD KV objects for its own metadata, although that was the appropriate design choice. Unfortunately, relying on an early and unstable version of MDHIM created performance bottlenecks and a large amount of debugging complexity. Even with an improved MDHIM, it might be beneficial to group multiple IOD KV objects into a single MDHIM table and then split as necessary (somewhat akin to the splitting decisions made by GIGA+). This bundling technique might be useful as well for all three object types using concepts and code engineered in PLFS small-file mode.

Finally, leveraging existing software provided both benefits and costs. The benefits of code sharing are obvious but the difficulty of adding multi-version control to software oblivious to it, such as PLFS and MDHIM, was possibly harder than writing the needed software from scratch, since multi-version control is such an all-permeating feature. In addition, there were several problems in particular with MDHIM and PBL-ISAM which illustrate the difficulty of reusing code in an environment and workload different from its original purpose. One difficulty was due to PBL-ISAM trying to read freshly written data in the course of a single persist operation when IOD asks MDHIM to save its tables to DAOS and MDHIM forwards the request internally to PBL-ISAM. The EFF stack does not allow reading from uncommitted transactions, so this PBL-ISAM behavior was incompatible; since the code was hidden deeply within MDHIM, it was both difficult to detect as well as to repair. Finally, MDHIM displayed unusual MPI behavior, combining threads and barriers in a way which had not previously been attempted in Cray's MPI and causing a very difficult search for performance slowdowns on the LANL test-bed system.

5.4 HDF5 & Mercury

HDF5 is the anticipated entry point to the EFF stack for many applications and domain-specific I/O libraries. Designing and providing an intuitive and high-performing interface to the new EFF capabilities (transactional I/O, BB space management, asynchronous operations, and end-to-end checksums), while retaining most of the existing HDF5 APIs and features, has posed several technical challenges.

Working with the ACG team to formulate a data model for their use case based on HDF5 and the EFF stack provided excellent feedback regarding features that were confusing (or just plain opaque) and brought a dose of reality with respect to what could be completed within the prototype project timeline versus what a production-ready EFF stack could enable. Porting VPIC-IO and H5Part, and adding query, indexing, and analysis shipping each stressed the entire EFF stack (software and architecture) in different ways.

5.4.1 User-Facing Findings

HDF5's traditional API was retained and extended to support new EFF features. This made ports of existing software (such as VPICIO and H5Part) to the stack more straightforward, and allowed the ACG team to begin the development of the HAL layer using traditional HDF5 files before the EFF storage layers were ready. While there were many advantages to this approach, the similarity of the calls at times obscured the fundamental differences in semantics. This was particularly apparent with write APIs, which do not immediately change container contents in EFF, but rather add updates to transactions that will only be visible after the transaction has been committed. Providing 'transactional wrappers' around the underlying HDF5 APIs that reinforce the notions of transaction updates and reads from container versions may help users adopt the appropriate mental model for EFF I/O.

HDF5 property lists, rather than multiple additional function parameters, control the checksum operations, specify replica IDs in read operations, pass number of leaders in transaction start, and support many other EFF-specific capabilities. While property lists offer a very extensible approach, they can also make important controls nearly invisible to the user and cumbersome to set. Higher-level interfaces to these setting should be provided in production releases, possibly resulting in more routines, but ones that are easier to understand and use.

The event stack extension to HDF5 allows an application to bundle multiple asynchronous operations and then easily test or wait on all of the events to complete. Unfortunately, when failures occur the application must parse through the operations in the stack individually and query each for success, extracting and re-executing operation(s) as needed. In future iterations of the event stack feature, there should be easier ways for applications to handle these situations.

The new HDF5 Map object provides users with a natural structure for storing related key-value pairs, and should make operations on this type of data much more intuitive than it was with the traditional HDF5 data model. The expectation is that the Map object will receive increasingly heavy use, especially if it can perform well on a wide range of key/value data types and sizes, and with both very small and very large entry counts.

5.4.2 Stack-Facing Findings

A significant portion of the changes to HDF5 involved instantiating the HDF5 data model on IOD objects and aligning operations expressed via the HDF5 API with their counterparts in IOD. The HDF5 Virtual Object Layer (VOL) capability enabled these tasks, using a VOL plugin tailored to this purpose.

Although the Solution Architecture stated that IOD calls would be shipped from the CN to the ION, this was a poor choice for two reasons. First, individual HDF5 operations frequently map to many IOD operations, and second, shipping IOD calls would complicate the asynchronous operation implementation. Consequently, HDF5 uses Mercury to ship HDF5 VOL calls from the CN to the ION, minimizing round-trips and allowing HDF5 to perform sequences of IOD operations for single HDF5 operations locally on the ION.

Traditionally HDF5 controls the allocation and distribution of IDs for objects in a container through collective operations. To enable independent object creation on each CN and avoid collectives, the IOD object ID range is partitioned by HDF5 at container open, with an equal

number of IDs allocated to each CN. The number of IOD IDs used by each CN is tracked and stored in the container when it is closed, so that object creations in subsequent operations on the container don't duplicate ID allocation. If this metadata is not found on container open, it will be reconstructed from the container contents, with some performance sacrifice. With the 62-bit address space currently available for the IOD object ID range[14] and on the order of one million CNs planned, this approach allows each CN to create approximately 2^{52} IDs. If this limit seems overly restrictive, the IOD object ID range should be increased.

Historically the HDF5 library had direct control over object layout and caching policies, and considerable effort was put into optimization of HDF5 metadata structures to enable fast lookup. The initial EFF approach to use individual IOD KV stores for the HDF5 per-object metadata information may need to be reconsidered to get the required performance, as small accesses to multiple KV stores may never perform well. Potentially hints regarding access patterns on KV create could optimize the representation used by lower levels of the stack.

Mapping HDF5 objects to IOD objects was straightforward in general, but some 'rough edges' were exposed that could not be fully resolved in the prototype. Careful consideration regarding which layer is responsible for tracking sets of related objects is warranted, with a combination of performance considerations and code reuse opportunities guiding the decisions.

Finally, the transaction model that IOD provides presented some challenges for the HDF5 layer:

- IOD provides a single transaction model for both writes and reads. A write transaction is started on a container that is open for write, updates are made, and the transaction is finished and committed. Each write transaction must be coordinated by participating ranks at the application level. The start of a read transaction on a container indicates the container contents as of that committed transaction must remain readable. Ranks are not allowed to independently start and finish read transactions; therefore if two ranks want to read from the same container contents, they must coordinate their read transaction start and finish.

  At the HDF5 level, different terminology and models for writes (transactions) and reads (container versions) have been adopted. On the read side, application ranks should be able to acquire an RC for a given CV without having to coordinate with each other. While coordination might improve performance, it should not be mandatory as the data being accessed will be consistent regardless of the operations by the individual ranks (or applications). With the current IOD implementation, it is not possible to provide this functionality; although the IOD API allows the capability, it is not guaranteed to work properly. This can be addressed in future work as IOD currently does reference counting on open objects so the API can be extended to ensure that re-opening already open objects is guaranteed.

- Each IOD object can be opened for read or for write, but not for read-write. The IOD object open operations, which take an IOD objectID and a transaction ID, return an object handle that is used in subsequent write or read operations.

[14] IOD object IDs are 64-bit values, but 2 bits are reserved for internal IOD use.

Opening the object for read or for write can give hints to the IOD library that could allow access optimizations. However, for the HDF5 library's usage of IOD, segregating the object open into distinct read and write parts proved to be very cumbersome. Writing to an HDF5 object involves both reading from that object and other objects associated with it to retrieve metadata information required to perform the actual write operation. This means that the HDF5 library is forced to always open each IOD object twice, once for read and once for write, and then to use each handle appropriately. This could be made simpler at the IOD level by providing a single API for opening objects, since accessing IOD objects already includes an IOD transaction ID that indicates whether the access is a read or write.

- While it is true that every IOD object may not be present at every state of the container, it is not clear that specifying a transaction ID in the IOD object open calls offers any real value in addressing the issue, since the returned object handle can subsequently be used with other transaction IDs. DAOS's model where every object (identified by object ID) always exists is worth considering at the IOD layer as well. Then, at the HDF5 layer, the IOD objects referenced would always exist, but may not be findable from the HDF5 container structure, and hence not be valid HDF5 objects. However, HDF5 is a bit of a special case since it maintains a lot of its own metadata about the contents of its data. Other applications using IOD which do not maintain such much metadata may want to be able to query an IOD container for which objects exist.

- IOD writes take a write object handle (WOH) and a write transaction ID (WTID), which maps well to the HDF5 interface.

  IOD reads take a read object handle (ROH) and a read transaction ID (RTID), which maps well to the HDF5 interface at first glance. Diving deeper, however, the RTID at the HDF5 level is a per-container indicator (the RC), while at the IOD level, the IOD library modifies the RTID in calls which make replicas, making it not per-container but per-object. In particular, IOD prefetch takes a ROH and a RTID and modifies the RTID to include a replica ID indicator. The modified RTID' is then passed to subsequent read and evict operations when the user wishes to work with the replica of the object data.

  Transparently modifying the RTID is problematic at the HDF5 layer, because the same RTID (RC) is also used to refer to a specific state (CV) of other objects in the container - not just to the object being prefetched. This "branching" of the RTID by the IOD layer makes it difficult for the upper layers to generically request data at a given CV. In addition, subverting the replica ID under the RTID makes it impossible for a single replica ID to be used as an identifier for a set of data that was prefetched together at the IOD level, and very difficult to build such a capability at the HDF5 layer.

  From the HDF5 perspective, IOD transaction IDs should be immutable, and read and evict operations should take a ROH, a RTID, and a replica ID, where the same replica ID may be used to 'color' multiple objects that should be acted on as a set. This would allow, for example, all objects under an HDF5 group to be prefetched together using the IOD's list operator, and then evicted with a single imagined IOD function: *evict_all_objs_matching(RTID, replica ID)*.

  The IOD read would take arguments ROH, RTID, and optionally replica ID. If no

replica ID is specified, IOD could find the 'closest' data. As further discussed in Section 7.2the team discussed the 'find the closest copy' capability (sometimes referred to as *tagless read)*, and it was generally felt to be a good idea, but there was insufficient time to implement the capability in the prototype phase.

- IOD offers a single API to evict data that is in the burst buffer as the result of write operations, and to evict data that is resident due to prefetches. The evict call takes an IOD object handle and a transaction ID, where the transaction ID may be a transaction ID' if the data was prefetched (or replicated from elsewhere in the BB, although the HDF API currently does not expose that capability).

  Based on past experience, The HDF Group felt that the ability to evict at the fine per-object per-CV granularity is valuable in order to allow applications to manage their BB data footprint. Having this capability in the current prototype provides the building blocks for larger-granularity burst buffer management in the future and gives the maximum degree of control in the present.

  When evicting replicas, there is no danger that any data will become unreadable. When evicting freshly written data, however, the readability of some data may be lost in ways which may be confusing to users of the IOD API. If for example, transactions 1 through 5 have been committed at the ION layer, they are all readable. However, if the user evicts any of the RC's, than RC's for some of the other RC's may also become unreadable. For example, if RC 3 is evicted, then it is not surprising that RC 3 is no longer available to be read; however, it might be a bit more surprising that RC 4 and RC 5 which were not evicted have also become unreadable. This is because some of the data needed for them may have been part of transaction 3 which created RC 3 and IOD removed that data when it evicted RC 3.

  This was a confusing semantic that evicting a particular RC could effectively render other RC's unreadable. At the HDF5 layer, a preferable semantic for evict would be to merely inform IOD that only that particular RC will no longer be read. An alternative operation at the IOD layer that would have been useful to HDF5 would be to ask IOD to preserve the readability of one particular RC and evict all older unnecessary data; an operation that was often referred to internally as flattening. The current IOD API allows an approximation of this in which a user creates a replica and then evicts the data from which the replica was created. However, due to the lack of tagless read as discussed above, this was not a viable option. This mismatch was discovered fairly late, as each team read and interpreted the design documents from their own mental model. Considerable effort went into adopting terminology that would allow the team as a whole to communicate, understand, and form consensus. While much progress was made, the BB data movement area is one where there is still considerable uncertainty about the best approach. Hopefully, when IOD adds tagless read as planned, this confusion will be at least somewhat resolved.

5.4.3 Legacy Capabilities

Certain aspects of the HDF5 data model do not fit well into the architecture and feature set of the EFF storage stack, and have been deferred for later implementation.

A significant capability mismatch is the iteration functionality that HDF5 provides for

various HDF5 objects. For example, iterating over all links in a group and making a callback to a user-provided function for each link. There are two primary difficulties with iteration operations in the EFF storage stack: (1) they cannot be executed asynchronously (since they call back into application code), and (2) each iteration callback is likely to require a round-trip communication between the CN and ION. The iteration functionality is most frequently used by applications when discovering the structure of the HDF5 Objects in the container through traversal. Potentially few HPC applications rely on this capability, as they typically know the structure of their files in advance. And, when structure discovery is used, performance is generally not critical. Support for traditional HDF5 iterators and/or another EFF-appropriate method for discovering and traversing objects in a container should be added, with capabilities and performance trade-offs driven by user community requirements.

Another aspect of the HDF5 data model that hasn't transitioned well to the EFF stack is support for data that varies in size. In particular, variable-length datatypes and extensible HDF5 datasets. Variable-length datatypes require two round-trips when accessing data elements, as Mercury requires the buffers for receiving data be allocated in advance. Because of this, the lengths of the elements must be sent first, then the actual element data. This may be acceptable for many HPC applications, as variable-length dataset elements are rarely used. Variable-length datatypes will be likely be more common for Map values, and the performance could be improved with enhancements to Mercury.

Extending dataset dimension sizes turns out to be quite difficult with the current EFF transaction model. If all application processes wish to extend the dataset dimension to the same size, all is well. However, if each process wishes to independently extend the dataset and append some data, a read-modify-write scenario is created for the dataset's dimension size information within the transaction; functionality that is not allowed in the current transaction model. The application could manage this information at its level, but then the dataset appends would not be independent. The best approach would be to have the HDF5 library provide an 'append' operation, built on top of an append capability in IOD. Initial work toward this functionality was undertaken in the project at the IOD level. However, the current IOD append feature does not support knowing where (i.e., at what offset within the object) appended data lands, and only works with blob objects. Possibly a blend of tracking the offset of appended data by the HDF5 library and the IOD append operation would solve this issue, but further thought and research is needed. For some use cases, the new HDF5 Map object might provide the application with the capabilities it needs to 'store *something* at a *place* where it can later be accessed', where the *place* would be an application-assigned Map key, and the *something* would be the Map value.

5.4.4 New Capabilities

The design and implementation of new data indexing and analysis capabilities in HDF5, first prototyped as part of the EFF stack, has also offered several lessons.

Although the capability to index data to enable future performance improvements for query operations is valuable, it comes at a price, both in terms of container size and time spent creating and maintaining the indices. Storing an index in a container has an unavoidable cost in terms of space. Asynchronous operations can help hide the performance impact of index creation from applications; alternatively, indices could be built and stored on objects after the application creating the container has finished writing to it.

An aspect of indexing that wasn't obvious prior to implementation was a limitation on

simultaneously updating indices from multiple application ranks during the same transaction. An index is a shared resource for each HDF5 dataset, something that each rank needs to modify when it writes data to the dataset. Each rank's operations are independent and uncoordinated by HDF5, which creates the potential for both a race condition on index modification and an invalid read-modify-write situation on the index within the transaction. Addressing these limitations appears to be possible, but would require a capability to cache the index's current state and to manage access to it, probably on the IONs.

Another limitation of index operations is the inability of FastBit and Alacrity to execute in parallel, on multiple nodes in the storage system. This restricts queries to sequential operation, although again, asynchronous execution mitigates the issue for applications. It has a larger impact for analysis shipping operations, since they are dependent on an initial query operation before executing the 'Combine' operations in parallel, and the application is likely waiting on the results.

The analysis shipping capabilities created as part of this project are also a source of implementation lessons. It was not obvious at design time that the analysis shipping capability would require a "co-resident" version of the EFF stack, one that directly translated HDF5 calls to IOD on IONs and SNs and that didn't require a function shipping infrastructure. However, due to the press of time to develop this prototype, a separate VOL plugin that bypassed Mercury was not developed. Instead, Mercury was enhanced with a co-resident capability that bypasses sending messages through the network abstraction layer, enabling the same HDF5 VOL plugin to be used for both normal CN-ION connections and to directly invoke HDF5 calls on IONs and SNs.

Several limitations exist in the current implementation of analysis shipping, mainly due to the experimental and in-flux nature of the feature, which reduced the time available to develop a seamless interface to the feature. Consequently, all the data within a container must be either located in the burst buffer, or on DAOS - the analysis shipping infrastructure currently doesn't have the capability to address containers that span both locations. Additionally, the application interface for locating the IONs or SNs where the analysis operations should be shipped is very limited - the node names are hard-coded and the application must know whether the container data is located on IONs or SNs. These do not appear to be intrinsic limits to the ease of use of the analysis shipping capability and further time spent polishing the feature would remove these restrictions. For example, the IOD API was improved at the end of the project to report the DAOS shard ID for data stored on DAOS as well as the rank of the IOD process closest to the data for both data stored on DAOS and in the IONs.

Integration of the new HDF5 library with Python using the h5py module highlighted the usability issues brought on with the introduction of transactions and read contexts. The new versions of the standard HDF5 functions required either a transaction or read context parameter, thus creating execution order dependency. Propagating the required changes through the h5py module's codebase proved challenging as the high-level interface heavily depends on the assumption that there is no such interdependency in execution between various HDF5 functions. Despite this, the modified h5py module does provide the level of functionality appropriate for the proof-of-concept nature of this project, although more structural changes will be required to bring it to a production status.

5.5 ACG

The key findings of the ACG group focus on making the compute layers more intuitive for graph analytics by including elements of the Map/Reduce paradigm and integrating the Hadoop framework into the architecture.

5.5.1 Difficulties of data ingest

Hadoop is the most common large scale data analytics system in the commercial world. Thus, it was a natural choice to leverage existing Map/Reduce packages and integrate them into the EFF stack. Hadoop differs from MPI-based systems in several ways. One important difference is the transient nature of the working processes, in contrast to the EFF stack, where processes are expected to be static. In Hadoop, management systems (e.g., Yarn) monitor the speed of processes and coordinate fault tolerance, restarting computations that have failed—sometimes moving them from one machine to another. Architecturally, this fluidity poses at least two challenges:

1. It is hard for an application to notify the EFF layer of the exact number of processes that will be communicating with the stack in course of the application's life.
2. The job manager can restart the same jobs based on various factors. However, the aborted, and then subsequently restarted jobs may have created/committed transactions already, which they will again try to commit/restart. Keeping track of such aborted processes and their transactions make the application-level transaction management very difficult.

In summary, Hadoop in the current form does not mix seamlessly with MPI, and support for speculative execution over MPI platform might be a field of further investigation.

With the above challenges in the background, two different options were investigated for the data ingest module:

- **Hadoop jobs directly communicate with EFF stack:** In this model, Hadoop processes would share an MPI-communicator. This is the most natural choice, architecturally, but quite difficult to implement in practice, for the two reasons stated above. Besides, disabling speculative execution in Hadoop would result in loss of its intrinsic fault-tolerance model.

- **Format conversion in multiple stages :** In this approach, Hadoop processes would first create their outputs either as HDF5 files on a POSIX file system or as a flat file on HDFS (Hadoop distributed file system). In the second stage the created output would be read and then written to the EFF stack by a converter application.

  In the case of HDF5 files as the intermediate format, these files would get converted by a (yet to be developed) HDF5-format conversion program. If available, such a conversion program would probably be quite useful, given that many scientific applications in the HPC world already have their datasets in HDF5 format, and

thereby a conversion tool could get them onboard very quickly.

If the intermediate format is a flat file in HDFS, the format converter then would work between HDFS and the EFF stack. A set of MPI-based concurrent processes would pick up files from HDFS already prepared by Map/Reduce jobs, and then write the data onto the EFF stack in HDF5 data model using the HAL.

For this project, the format conversion approach was chosen, with flat files on HDFS as the intermediate data format. A bridge tool was developed by the ACG team to perform the final task of data loading onto the EFF stack via the HAL.

5.5.2 Data-crunching needs scalable-appends.

Data in the real world is rarely available in the format needed for all steps in the analytical pipeline. Thus, efficient pre-processing is not a secondary task. In fact, a proportionally large amount of time is spent in this step of analytics. In this stage, it is difficult—if not impossible—to know *a priori* the size of the data structures that will result from pre-processing. For example, while parsing a large, raw textual corpus, co-occurrences of words become independent edges, and there is no way of knowing the exact count of such edges without fully processing the entire corpus at least once. It is therefore important to have the ability to dynamically grow the data structures resulting from pre-processing; this growth should be scalable and supported for concurrency. While initial support for this capability was added in IOD during the project, there was not time to integrate, test, and deploy the capability at the HDF5 level where ACG would have had access to it.

5.5.3 Power-law in natural datasets imposes significant challenge in designing flexible data-structures.

Natural graphs tend to follow power-law degree distributions. For such graphs, it is quite challenging to pre-arrange a storage layout, as a global knowledge of the graph topology is essential to optimally partition the graph. One option is to pre-allocate space for larger parts of the data sets. Complicating this approach, however, is that power law-distributed graphs, by definition, only have a small proportion of vertices that would require this, and there is no way to know for which vertex this will be necessary without either first making a full pass through the data, or initially allocating a large amount of space for every vertex, resulting in a huge waste of space.

On the other hand, allocating blocks of moderate size and extending their count on demand would mean multi-level indexing and hence multiple round-trips for metadata collection to reach any specific part of the container. Clearly that results in very poor retrieval latencies. It also turns out to be a quite general phenomenon – in the sense that some parts of the datasets need to be given exponentially more resolution than others. An efficient solution would also cater to problems that are completely unrelated to graphs as well.

5.5.4 User-centric Pros/Cons of the Transactional Model

Pros - Transactions naturally support multi-version concurrency control. The experience gained using Transactions with graph-based computational kernels bolsters confidence that this is a useful model to implement multi-phase Gather-Apply-Scatter (GAS) style graph computation. Typically, in a GAS computation, some of the vertices are replicated on all partitions. One replica is called the master, and the others are shadows. All cross-node communication happens through these replicas, i.e. at periodic intervals the master and shadow vertices communicate their respective state to exchange information. As indicated in Figure 20 below (master indicated by blue, and shadow by gray, disks stand for a process running on a graph-vertex, and the squares for the information they read/write from/to the storage). The whole process of master-shadow reconciliation can be done in two phases. In phase-1 (transaction T1) the master and the shadow commit their copies to the file system, and then in the second phase, the master reconciles the two pieces of information and commits a second transaction (T2) that is read by all the shadows in the next phase. Without the transactional model, the application will have to write its own concurrency control, which is quite painful. Although the span of this project did not provide enough time to explore this model in detail, the semantics of the transactional model makes it clear that this could provide a natural support for asynchronous graph computation.

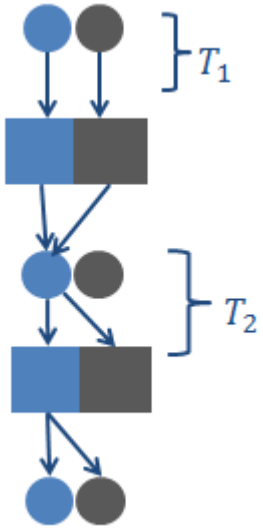

Figure 20: Master Shadow Reconciliation

Cons – Difficulty of developing middleware for the EFF stack. While trying to build ACG-applications, the goal was to keep the graph-computation logic separate from I/O handling. The next logical step was to put the I/O interaction inside a separate module which hides the architectural elements of the HDF5 constructs (and those of the EFF stack) from the application. As a result, the HDF5 adaptation-layer (the HAL) had to bear the responsibility of managing transactions on behalf of an ACG-application. But then, the sequencing of transactions being left outside of the EFF stack layers, and without specific help from the

application, it becomes tricky to assign transaction numbers to the updates performed by the HAL. This would not be an issue if there was a global scalable service that could provide assigned transaction numbers in a conflict-free manner. However, that was a design element beyond the scope of this project. The only choice for a middleware is to impose a specific sequence of transactions - a sequencing scheme which may not be optimal for the application in terms of the ordering of commits as well as their granularity.

6 Methodology

6.1 Overview

The FastForward Storage and I/O program was a Software Research and Development project to advance the DOE Exascale roadmap in I/O throughput and enhanced storage functionality, scalability and fault tolerance. Program development ran from July 2012 through June 2014 for a total of eight quarters. The output of this program was to develop a working prototype and demonstrate the utility of the I/O stack for Big Data by using the stack to create and store Arbitrarily Connected Graphs (ACGs) and run graph computations.

The software stack software engaged five distinct teams:

- BigData-HPC Bridge – ACG Application testing – Intel Labs
- Application I/O – The HDF Group
- I/O Dispatcher – EMC Corporation
- Distributed Application Object Storage – Intel High Performance Data Division (HPDD) and Data Direct Networks (DDN)
- Scale Testing – Cray

The skill sets and talent was specifically sought out and led to a remotely distributed team spread across a broad geography on three continents. During development, integration, and testing this provided the benefit of working around the clock. The downside to this arrangement was the challenge for debugging sessions across such a span of time zones.

Team engagement was high. This is a passionate topic for all members involved. While focus was placed on clear communication between everyone regarding layers of the stack, communication challenges were likely inevitable for the two shared test systems, which represented different architectures, during the concurrent deployment, debugging, evaluation, and reinstallation of new code during the integration and testing quarters of the program.

The increase from petascale to exascale for storage and I/O has required a new approach to the architecture because it is not possible to simply scale past systems. A new vocabulary of data structures and terms has accompanied this new architecture. During the course of the program, the team as well as the group of DOE stakeholders worked to arrive at a common vernacular. In some cases, this provided a deeper understanding of the architecture in question. In other cases, this hindered development efficiency due to the regrouping required to return to a common baseline.

Overall, the in-depth discussions surrounding the education of teammates and stakeholders produced a stronger design and prototype implementation as well as shaking out more corner cases and edge conditions.

The collective volume of related experience within the development team was apparent throughout the program as stack layers and approaches were explained internally and externally. The combination of team size and very aggressive schedule created a team in which the timely contribution of every member was vital. This was made apparent whenever members were not immediately available due to planned or unplanned work absences.

The compressed schedule also impacted the available time for fully benchmarking the stack layers. The team is continuing to run performance tests on the higher layers through the end of the final program quarter.

The goal for this FastForward-Storage and I/O program was to develop a prototype. Many of the decisions made along the development path were made with all available information at the time. Program documents, including this final report, explore the pros and cons of the design and architectural decisions made.

6.2 Processes

With a development team distributed internationally as shown in Figure 21 below, consistent twice weekly development meetings were held to discuss:

- Milestone scheduling
- Technical development and architecture of features and integration
- Preparation of software demonstrations
- Test cluster setup and usage
- Document preparation

Whenever possible, team representatives would meet in person following customer meetings or industry conferences to consolidate travel expenses. These in-person meetings occurred about once per quarter.

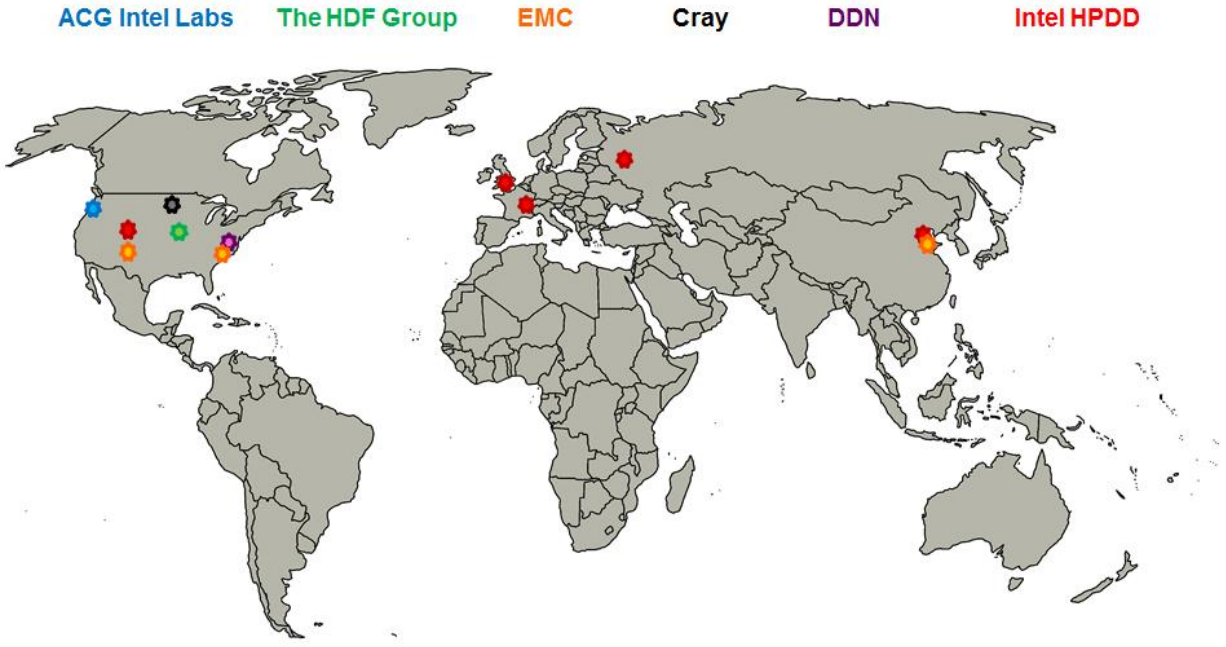

Figure 21: Global Team Distribution

By simultaneously designing and developing efforts at all levels of the I/O stack, the project participants became an integrated "I/O Co-Design" Team. Prototypes of both DOE Applications as well as ACG allowed users to experiment and guide the project. These also appropriately stressed the prototype storage layers of IOD and DAOS. A significant amount of learning and feedback occurred between teams.

Quarterly milestones were demonstrated to stakeholders in person when possible, timed with semiannual DOE meetings. Software demonstrations were otherwise held via remote conferencing sessions. A weekly telecon was held between program management and the contract and technical customer representatives. Signoff of quarterly milestones was requested after the conclusion of software demonstrations and delivery of accompanying documentation.

The quarterly software demonstrations were preceded by a high level review two quarters in advance and a detailed level review one quarter in advance. This allowed engineers and stakeholders to ensure a common understanding of what would be delivered. This cadence was a successful risk mitigation approach for the majority of milestones. However, both engineers and stakeholders reflected that it was difficult at times to think that far in advance, especially at the beginning before the demonstrations included integrated software layers.

The team presented to the DOE twice annually at LLNL during DOE program reviews. At these meetings the team would discuss the progress made over the previous six months and present the next 3-6 months of planned efforts. These meetings were also an opportunity to present the necessity of the fundamental change in storage and I/O architecture to a representation of the larger DOE application scientist body, fielding their questions and concerns as well as receiving their input as the end users of a final product.

6.3 Test Resources

Intel's Lola cluster and the ACG cluster have similar configurations. Both of these systems provided useful locations for development, integration, and testing. The third test cluster was a Cray system set up at LANL with a dedicated system administrator. Having a dedicated system administrator on the Cray test cluster was beneficial both to have a third party perform the installation of the EFF stack for the first time as well as shoulder the effort of system setup and maintenance, which offloaded the management of system complexity from a development perspective. It would have been beneficial to have system administrator help with the Lola and ACG test clusters as well to be more efficient in the development process. While navigating issues on multiple systems diverted time from performance tuning and development/testing, it was beneficial to run on multiple systems to show the portability of the EFF stack.

6.3.1 Intel's Lola Cluster

The Lola cluster, installed in Intel's premises, was made available to the Fast Forward team early in the project. The cluster is composed of 27 Xeon DP Sandy Bridge E5-2680 nodes used as follows:

- 1 management node which is also the chief server
- 6 storage nodes, each connected to a private 45-disk LSI JBOD. Each node is configured as Lustre OSS with 2 OSTs using RAIDZ over the JBOD disks.
- 4 storage nodes connected to a Netapp appliance used as OSSs and MDSs.

- 8 I/O nodes with 8 Intel Ramsdale SSD disks each. A Lustre ldiskfs-based file system has been created over the 8 local SSDs on each ION which uses this file system as local burst buffer. Cross-mounts are set up to allow all bust buffers to be accessible on all the IONs.
- 8 compute nodes where HDF5 applications are run.

In the first part of the project, the Lola cluster was used to develop and test DAOS. Its integration into the Lustre test infrastructure has been very helpful to allow new DAOS builds to be deployed and tested quickly.

The Lola cluster was later used for the EFF stack integration. The flexibility to install new software and the cluster's ease of access for all team members have been key factors to a successful collaboration.

6.3.2 Intel's ACG Cluster

The ACG cluster is composed of 32 Xeon DP Servers (Intel Grizzly Pass 2U) and two Engineering Servers designated for cluster management and development/build. Details of configuration are as follows:

- 16-node multi-purpose cluster (Xeon DP Servers) configured with Hadoop/HDFS and DAOS POSIX based EFF stack. DAOS POSIX based EFF stack has been used for early FastForward development. Each server is equipped with six 4TB HDDs for local storage and HDFS.
- 8 IONs configured with DAOS Lustre (Xeon DP Servers). Local SSDs are configured as burst buffers. Each server is equipped with four Intel 910 series 400GB SSDs for burst buffer.
- 8 CNs (Xeon DP Servers) with GraphLab, Hadoop and HDFS, HAL, and HDF5. Analytics applications load data either from HDFS or EFF stack to run computation. Each server is equipped with six 4TB HDDs for local storage and HDFS.

The ACG team split the 32-node cluster into two 16-node clusters: one for early development and another for full EFF stack integration with ACG applications. The early development cluster has been configured with GraphLab, Hadoop/HDFS, HAL, HDF5, and DAOS POSIX. With this setup, the team was able to develop ACG applications and HAL without relying on DAOS Lustre development schedule. The second half of the cluster, then, was configured with DAOS Lustre for full stack integration, test, and demonstration.

6.3.3 LANL's Buffy Cluster

"Buffy" is set of systems based around a Cray XC-30 with 64 compute nodes and 14 I/O nodes. The other major components are the "Lustre File System by Cray" (CLFS), and EMC's "Active Burst Buffer Appliance" (BB) system. The CLFS is a Lustre appliance system using a DDN SFA12K disk array, with 4 OSS's and 2 MDS/MGS systems. The Burst Buffer is comprised of 3 (of 4) systems, each serving Lustre 2.4 to the IONs. Each of the filesystems is composed of 1 MDT and 7 OSTs.

Porting the FF-Storage stack onto the Cray architecture proved to be challenging since:

- CentOS was used on the IONs, with the compute nodes remaining with SuSE's OS. While beneficial in being able to use a supported OS for DAOS clients, it made the user's environment more complex by having to compile and run on two OSs

- Lessons learned on the Intel Lola cluster did not always transfer directly to the Cray, for instance what worked for compiling the stack on Lola did not work on buffy
- The security infrastructure, as well as some late network connections directly to the login node, created impedance that caused developers to work where they felt most productive, namely on Lola
- The multi-threaded MDHIM implementation does barriers within threads which is an MPI pattern problematic for the Nemesis progress engine running within the MPI provided by Cray to add features needed by EFF and caused tens of seconds latency for some simple IOD operations such as transaction finish and container open which use MDHIM internally for the IOD metadata

The challenges, although mostly overcome, came at a price of having not having sufficient time for performance testing and running a majority of demos on the Cray.

6.4 Recommendations for Future Projects

In several of the quarters, new code functionality was delivered that required dependent code from across the stack to be delivered within the same time frame. This reduced the test and debugging time for lower layers and also reduced development and testing for higher layers. The future recommendation would be to build the quarterly delivery schedule for milestones that have a high level of dependency so the required functionality in the lower layers would be completed and tested in the previous quarter before higher layers required this functionality to be in place and fairly stable.

Integration of the software was scheduled for Project Quarters 6, 7, and 8. This was likely unavoidable as the groundwork needed to be carefully constructed, however it caused the intensity of the final three quarters to be significantly increased over the earlier portion of the program. The takeaway lesson would be to limit new feature development during integration quarters to keep a steady team workload.

Two additional process lessons are recommended for similar projects in the future. First, it is recommended that all layers participate in the development and maintenance of a continuously evolving regression test suite. This will require additional project scoping as the effort to do this is significant. It is equally important that the regression test suite be trimmed of outdated tests and kept updated to test new features to ensure the results are reasonable in both time and space to reduce the search space for any newly introduced problems. All layers need to ensure that new code delivered passes the regression test suite. Although this was attempted in a somewhat ad hoc fashion, the failure to do this officially and thoroughly cost much lost time when newly delivered code would break old code. Second, delaying performance testing until all functionality was delivered allowed the possibility for performance faults to become more deeply embedded. In future projects, performance metrics should be designed and delivered earlier in the project.

The installation learning curve was steep for the new test systems in place. It is recommended that future test system environments be set up two quarters in advance of requiring them for demonstration milestones and plan internal usage by developers one quarter in advance. Benchmarking the system before team usage and stack installation would have been beneficial as a point of comparison for performance debugging. In this program, the team would have benefitted from a more detailed set of instructions for logging in, building code, and running MPI jobs.

It would also have simplified usage of the Cray system for users if a single OS was set up and a future recommendation would be to investigate running SuSE on the IONs, for example. This is primarily a task of verifying that DAOS client code work on the ION, and fixing any problems that are found.

7 Future Work

It is unfortunate that time pressure at the end of this project meant there was very limited time for performance testing both with synthetic benchmarks and with real applications. The LLNL IOR benchmark was ported to run directly on top of the HDF5 API, the IOD API and the DAOS API. The LANL fs_test was also ported to work with IOD. These proved extremely useful and fully characterizing the performance of the prototype would enable a much more detailed understanding of the benefits and drawbacks of the architecture. Insight gained could guide the adjustments, refactoring and tuning required to make the EFF stack perform to its best under a wider range of workloads.

Close cooperation with DOE application developers to port their applications during this characterization process would be highly desirable. There was less project scope available for this since the prototype functionality was very limited in early stages, however all the inputs from application developers received during the project were extremely valuable and this would be expected only to increase now that the stack's main functionality can be demonstrated.

The transaction model merits further development. As mentioned in the technical findings, POSIX developers find programming in an environment in which data becomes readable only after commit quite unnatural. In some respects this project attempted to see how far that model of I/O could be taken and have prototyped it. However, some relaxation, for example to provide visibility of one's own uncommitted updates, may lead to an easier transition for application and middleware developers. Storage class memory could even make it feasible to abandon this rule altogether since storage latency crucially determines whether the integration of versioned updates can be done on demand or must be batched into commits. Further investigation should be a fruitful avenue to pursue.

A major requirement for exascale storage, unfortunately not developed or demonstrated in the prototype, was replication to improve data durability and availability. It *was* one of the key features that the EFF transaction model aimed to support, since consistency must be guaranteed between distributed erasure-coded replicas to ensure that data is reconstituted accurately when a member of a redundancy group is lost. IOD is the natural choice of level in the EFF stack to add this facility since it already manages sharding over DAOS objects and it may in fact be worth considering splitting IOD into separate burst buffer management and sharding/HA layers as a result. Since time to repair determines the level of redundancy and therefore space efficiency and cost of storage, the major challenge will be to enable the recovery of lost redundancy during concurrent write access. This will require the stack level responsible for HA to coordinate with potentially conflicting instances of itself, which may require the underlying stack levels to provide new conflict avoidance services.

The EFF stack uses MPI both in Mercury, as a fast, low latency transport to allow the EFF stack to span different node types, and in IOD, to distribute its metadata via MDHIM. One unfortunate feature of MPI is that most implementations poll by default for completion in the interest of lower latency. There is no standard way to control this and some

implementations do not support blocking at all. This may be irrelevant for HPC applications which can rely on dedicated cores, but it creates problems for I/O middleware running on IONs with oversubscribed cores. Further work on the EFF stack would greatly benefit if a standard was established in the MPI community to enable control of polling, such as facilitated by Mercury, to promote wider adoption beyond the EFF stack.

POSIX support may seem a strange thing for an I/O stack that aims to replace it, but it is clearly required since it is foundational to practically all current HPC applications, many of which will have lifetimes well into the exascale era. The approach taken in the project, to add DAOS support to Lustre, an existing POSIX filesystem, made it natural to simply allow POSIX and the EFF stack to run side by side, exploiting the same function shipping, but bypassing the other EFF stack components. Future implementations could reconsider this, for example by providing POSIX namespaces layered over IOD encapsulated in their own DAOS containers. Certainly, interfaces to exchange data with the POSIX world will be required to enable collaboration between scientists working at different sites and institutions and POSIX support will continue to be required for executables, libraries, scripts and configuration files in any case.

One of the motivations for placing burst buffers on independent nodes in the system (i.e. not on the CN's nor on the OSS's) was to maintain separate failure domains. Note however that this assumes a fault resiliency which does not yet exist at the application or run-time level. Currently the Mercury servers and clients cannot survive any failures. However, the benefit of independent failure domains requires that the CN layer and the BB layer can be reconnected after a failure. This work will need to be done in Mercury and/or in a fault-resilient MPI. Additionally, it was explicitly outside of the scope of this project for BB data to be resilient across crashes. Although IOD, in some instances, can recover data in the BB's (assuming no BB data was lost), it was not designed to do so nor tested.

One key feature of the EFF stack that has not yet received sufficient attention is the ability for job data to be prefetched into the BB's before the job itself is scheduled on the CNs. Future work to re-invent data declaration languages like the JCL is needed to return this functionality and extend it appropriately for the exascale and integration into HPC schedulers.

The following sections describe further work required at each stack level to take the EFF stack forward into production. *All* stack components will require a program of development to address gaps left due to expedience during prototype development. A program of at-scale test and stabilization undertaken in partnership with the DOE, starting well before the software is deployed in production will also be required. Based on previous experience, both programs will be essential to deliver stable high performance storage software.

7.1 DAOS

Further work can be divided into three main areas. The first is to productize the DAOS/Lustre prototype to improve performance and stability, the second is functionality changes and additions to integrate better with the rest of the stack and remove dependency on the Lustre MDS for container metadata and the third area anticipates future HPC cluster architectures featuring ubiquitous NVRAM with an alternate DAOS implementation suitable for Storage Class Memory – DAOS-M.

7.1.1 Prototype Productization

The DAOS client's writeback cache requires further integration with the Linux VM subsystem to ensure that the cache reduces its memory consumption under memory pressure. Kernel metadata used to maintain the writeback cache should be refactored to maintain epoch-extent validity for read in addition to write to support read-ahead. Further development to support lockless cache operation to track validity as the HCE advances could be an interesting area to investigate.

Although container shard read and write exploit the existing Lustre event driven I/O dispatcher, other operations such as container metadata operations are dispatched to a dedicated kernel thread pool which sends blocking RPCs. These operations are therefore serialized once all threads are busy and this code should be refactored.

Further stabilization and performance work is also required on the Versioning Object Storage Device. Commit latency determines the natural transaction size and improvements here would support a wider range of workloads.

7.1.2 Functionality

Changes to the DAOS API to converge on a transaction model more consistent with the rest of the EFF stack were discussed in the technical findings. This should include additional features in DAOS to describe specific dependencies and support shared write access by coordinated applications to enable transparent background reorganization and redistribution of container data. Additional DAOS primitives should also be considered. Key-value objects would have greatly simplified aspects of the IOD implementation as described in the technical findings. Improved checksum integration with the rest of the EFF stack may also be beneficial.

Container metadata is stored entirely on the Lustre MDS in the EFF stack prototype. This relies entirely on MDS resilience to guarantee data durability. An alternative would be to replicate this metadata using PAXOS[15] over a sufficient subset of container shards. This would improve scalability for concurrent operation on different containers. A further improvement would be to implement the POSIX namespace currently hosted by the MDS within a DAOS container. The consistency guarantees provided by the EFF stack make the implementation of a scalable distributed namespace considerably easier and, through replication, more robust than the current Lustre implementation. The MDS could then be removed entirely from the storage system.

All storage systems require management tools and storage systems implementing DAOS are no exception. Further work is required to extend DAOS to include instrumentation and control APIs that facilitate provisioning, monitoring and management. New I/O APIs leveraging DAOS's underlying versioned object storage are also required to enable incremental container replication and migration to facilitate tiered storage management,

[15] Leslie Lamport, *The Part-Time Parliament*, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169

backup and recovery. A project to deliver a standard, scalable toolset built using these extensions that could then form a foundation on which storage system vendors can build differentiated management products.

7.1.3 DAOS-M

Future cluster architectures are envisaged based on compute nodes with integrated ultra-low latency fabrics that spread low powered NVRAM throughout the fabric. This distributed Storage Class Memory will be remotely accessible at the full cross-sectional bandwidth of the fabric and deliver bandwidth three orders of magnitude higher than disk based storage at latencies measured in processor cycles rather than 10s of milliseconds. These architectures will naturally support Distributed Persistent Memory programming models which have the potential to revolutionize how technical computing workflows are constructed and scheduled.

Although DAOS as an abstraction extends to these new architectures, they will demand a new DAOS implementation, DAOS-M that scales beyond the current Lustre-based implementation to hundreds of thousands of storage nodes and is extremely lightweight and tightly integrated with the fabric to ensure application I/O benefits from the fine granularity and low latency of these new technologies. Memory mapping extensions to the EFF stack could then support higher-level Persistent Memory programming models and interface to workflow management systems to control migration of Distributed Persistent Memory images both within the compute cluster and between the compute cluster and permanent durable storage.

7.2 IOD

An early design choice in IOD was to embrace shallow fast progress instead of deep slow progress by which philosophy IOD embraced the relaxed requirements of prototype software in order to attempt to design and explore as many high-level API features as possible. Through this decision, a tremendous amount of information about the abilities of exascale storage as well as user needs of it was gained. The trade-off is that some of these features need more testing before they are ready for production use.

Additionally, external libraries were often leveraged because they provided the needed functionality even though it was understood that they were incompatible with the eventual design. Specifically, the IOD processes currently fetch data from remote burst buffers by cross-mounting all burst buffers as Lustre file systems. This allows rapid deployment and development of upper-layer features but an N-squared set of Lustre cross-mounts in which each BB mounts every other is clearly not scalable. These should be replaced with a thinner IO forwarding system such as IOFSL or via MPI messages in which only the local IOD process access its local burst buffer. In the latter system, each IOD process needing remote data would send an unexpected message to its IOD sibling running on the target burst buffer node.

Similarly, IOD used MDHIM/PBL-ISAM as its key-value store but this choice proved regrettable both due to incompatibilities with transactions as well as being an unstable early version of the code. This layer needs to be replaced either with the new version of MDHIM or with a future version of DAOS which exports native KV objects.

Using MDHIM for KV stores also introduced a fair bit of complexity since it fundamentally changes how IOD buffers data in the burst buffers. Using PLFS, IOD logs all data for array and blob objects on the local burst buffer. MDHIM however does not do local logging; instead inserts into MDHIM are sent to the appropriate range server which might be running on a remote burst buffer. To improve performance, either the MDHIM client or the local IOD instance should buffer these local MDHIM inserts so that not every MDHIM insert need incur a ION round trip.

Additionally, due to the fantastic ease of using MDHIM, IOD currently stores its own metadata into MDHIM. Even if MDHIM however were a perfect layer of software, this introduces an extra unnecessary lookup. The choice was appropriate at the time to enable shallow fast exploration, but a better long-term design would be for IOD to store metadata directly itself using object and container leaders at the ION layer and using the metadata virtual container shards at the DAOS layer.

More work needs to be done to explore whether automated tiering is valuable instead of the current design decision to force users to explicitly control what data is stored in the burst buffers. The IOD team has worked with Lustre 2.5 HSM, Grau Data's PDM, and CEA's Robin Hood, so this work should attempt to integrate the automated scheduling and data movement in those systems into IOD.

Making efficient use of PLFS to store burst buffer data, IOD however does not currently use some of the collective optimizations found in the PLFS ROMIO ADIO module such as collective aggregation of PLFS metadata. In addition to this, IOD may benefit from recent work at LANL using MDHIM as the PLFS metadata manager or other research into PLFS metadata such as Jun He's compression work[16].

EMC's recent acquisition of DSSD foretells extremely exciting innovations in HPC burst buffer hardware. It is clear that IOD needs to be agile enough to allow a wide range of underlying hardware as well as underlying storage interfaces. Just as IOD today can access storage via either the POSIX or the DAOS API's, tomorrow's IOD may need to access direct KV interfaces to storage such as DSSD or Seagate's Kinetic Open Storage or to native DAOS KV objects should they be developed.

The HDF Group provided many valuable recommendations for making IOD be closer to the needed user programming modeling. Although many of these were followed during the course of the project, such as atomically appendable blob objects, many others were not possible given time constraints. These include the ability to create multiple replicas and issue reads on the original CV and allow IOD to find the "best" replica from which to fetch the data; in contrast, IOD currently requires the user to specify which replica in the read API. Also, the atomic append function should be extended to arrays and it should be implemented internally in a more scalable manner. Currently atomic append is done by querying the IOD object leader for a unique byte range at the moment of the append but a more scalable implementation would be to delay this until the commit of all of the appends.

---

[16] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun, "I/O Acceleration with Pattern Detection", in Proc. of the 22th International ACM Symposium on High Performance Distributed Computing (HPDC'13), New York City, NY, June 2013. http://pages.cs.wisc.edu/~jhe/hpdc2013-io-pattern-junhe.pdf

Although required for scalability at the exascale, asynchronous API's introduce new challenges which have not yet been solved in the EFF stack. One such challenge is that some errors may not be reported until a time significantly later than the function was originally issued. Although this is handled today by merely polling the status of the original request, in the EFF stack this is more challenging since many asynchronous requests can be merged into a single commit operation.

Although developed in the DAOS layer, scalable collective trees have not yet been implemented in the IOD layer. Since many operations, such as persist and fetch and replicate, can be issued to a single IOD process but are executed by all processes, IOD often needs collective communications but MPI does not currently allow an unexpected broadcast from an unknown root so IOD will have to implement this itself in a manner similar to the DAOS scalable collective routines.

At least two additional improvements have been identified, but not yet implemented, in the IOD-DAOS interactions. Both were identified in the IOD persist operation: an operation in which the IOD caller makes one single request to IOD which in turn becomes a very large number of requests from IOD to DAOS. One, IOD uses thread pools to improve its parallel access to DAOS. Using the DAOS asynchronous routines however would be preferable as the DAOS in-kernel thread pool can operate at lower latency than the user-space IOD thread pool. Two, it is possible that transient failures during persist may cause a large amount of redundant work. As a specific example, IOD may start a DAOS epoch, write a huge amount of data, and then encounter a transient failure at the epoch commit. Currently, IOD returns this failure to the application which can then re-issue the persist request at which point IOD will restart the epoch and resend all of the data. However, if instead of failing the entire persist operation, only the commit could be retried would allow the transient error to be resolved without resending all of the data.

Also deserving of more research and needing feedback from computational scientists is the best interface for reading or fetching large numbers of key-values in one operation. One possibility is to specify the first key to be fetched by its index into the global sorted object (i.e. fetch the next M key-values starting at the Nth key). The second is to specify the first key to be fetched using an actual key (i.e. fetch the next M key-values starting at key K). Note that the current version of MDHIM supports range queries but the older version used in IOD does not.

Finally, IOD does not currently checksum its own metadata and not all IOD functions, such as fetch, have list variants.

7.3 HDF5 & Mercury

Located at the top of the EFF software stack, future work for HDF5 and Mercury encompasses improvements for interactions with applications, tools to work with EFF containers, and enhanced features and better integration with lower levels of the stack.

7.3.1 Application Interaction

As the EFF stack moves toward stability and production status, greater emphasis should be placed on reaching out to the application communities that will be future users of the storage system. Exemplars that typify categories of applications (AMR, regular and irregular mesh storage, etc) should be targeted for conversion to the EFF storage stack.

These applications will provide a fertile ground for feedback about the performance, usability and capabilities of the stack. Time spent listening to and working with the application teams will be well worth the investment.

In addition to working directly with application teams, other higher-level I/O libraries, such as netCDF-4 or h5part built on HDF5 and heavily used by DOE applications will have to be converted to the EFF storage stack. Working with the development teams to create versions of these libraries for the EFF stack will enable higher-levels of abstraction for applications and speed adoption of EFF features.

7.3.2 Storage Container Tools

Tools that work with HDF5 data stored in EFF storage containers are necessary, both to enable scripted manipulations of container data and to leverage the existing infrastructure of analysis and visualization tools. Tools that can convert existing HDF5 data files to/from the storage container form will allow HDF5 data to be migrated between the EFF storage stack and other environments, likely legacy POSIX-based file systems. In addition, the existing HDF5 tool chain should be converted to access data in EFF containers, allowing application developers to easily compare, dump to text and copy data in the containers.

Existing visualization and analysis software used by DOE application teams, such as VisIt, Paraview and Ensight, should be ported to use the data stored in EFF containers as well. Without such analysis tools, there will be little reason for storing the data quickly and safely with the EFF stack. Fortunately, nearly all analysis tools have a modular framework for accessing data and plugins designed specifically for data in EFF containers can be developed without modifying the core infrastructure of the tools. In many cases, these tools have capabilities to query and extract stored data, which should be connected to the matching query and view interfaces in HDF5.

7.3.3 Enhancements to HDF5 and Mercury

Each of the features added to HDF5 and Mercury during the prototype phase has room for improvement, and many of those ideas are outlined here:

- Asynchronous I/O operations execute simultaneously on the high-speed interconnect with application communication, potentially introducing unwanted jitter in compute operations. The impact of asynchronous I/O should be measured and if necessary, a quality-of-service mechanism for interconnect use should be put in place, to segregate I/O and application traffic from each other.
- The implementation of asynchronous operations in HDF5 should be pipelined, so that asynchronously executed operations are allowed to execute in smaller stages, allowing more parallelism within the storage software stack and making broader use of the IOD and DAOS layers' concurrency capabilities. This would entail using IOD routines asynchronously and breaking up each HDF5 operation on the ION into a small state machine, managed by the task dependencies provided by AXE.
- The existing capabilities that IOD provides should be leveraged more fully: the HDF5 API should provide access to more IOD layout hints and the HDF5 IOD VOL plugin should leverage IOD list capabilities wherever possible.
- The transaction and container version model needs to be clarified and aligned fully with the IOD and DAOS transaction models. Ideally, a completely coherent transactional model can be settled on, one that provides both fault-tolerance and

ease of use for application developers.

- The capabilities of the burst buffer, or storage class memory, need to be fully exposed and exploited by the HDF5 interface. The current interface does not provide enough capabilities for exploring what data is located near the compute node and for leveraging that knowledge.
- Natural ways to specify sets of objects or sub-objects that should be prefetched and evicted as a whole should be added to the HDF5 (and IOD) interface. Allowing applications to specify these sets will greatly reduce the application management of objects in the container.
- Rough edges in the HDF5 API that deal with transaction and container versions should be polished and refined. Moving these important concepts into first-class position as parameters of all relevant routines instead of putting them in property lists would make the HDF5 API clearer and easier for application developers to use. Additionally, adding higher-level wrappers that encapsulated transactional concepts would simplify transaction operations.
- High-performance support for storing variable-length datatypes in HDF5 containers should be added, through improvements to Mercury that allow buffers of unknown size to be transferred efficiently. This will enable greater use of strings and sequences of data in applications similar to the ACG prototype.
- A capability for quickly and conveniently appending elements to HDF5 datasets should be added. This will require extensions to both the HDF5 component of the Mercury server on the ION, and new capabilities to append elements to IOD arrays.
- The new Map objects in HDF5 should have their feature set fleshed out, allowing each key and value to have a different datatype from others, implementing iteration over the keys/values and adding them to the indexing framework for queries to leverage.
- Capabilities for self-discovery of container and object contents should be created, either through implementation of the existing iteration routines in an exascale-friendly manner, or through other new methods for tracking and querying the structure of a container (or possibly both, since they might have non-overlapping use cases). One possible new method for tracking the structure of a container would be to store the entire hierarchy of the HDF5 graph as it is created and modified in a single IOD KV store. This would provide a central repository for all metadata about a container and could be used to avoid traversing the group hierarchy to locate objects as well as quickly answer global questions about the container.
- Similar to container self-discovery mechanisms, a way for analysis applications to be notified of new container versions appearing would greatly enhance their ability to avoid polling on containers to detect new versions and provide low-latency feedback of changes to the container.
- Mercury's capabilities should be expanded in several ways:
  - Being able to specify the transfer block size, which might be a subset of the user's declared total data size, over the wire would enable reading and writing large amounts of data at once (such as for checkpoint and restart operations), so that an application doesn't have to subdivide their buffers and make multiple HDF5 calls.
  - The server component of Mercury should be enhanced to allow transparent restarts of servers without loss of progress, in the event of failures.
  - Additional network abstraction layer drivers should be written, to take advantage of current and future high-speed interconnects.
- The new query, view and index capabilities in HDF5 have many areas that need polishing:

- The indexing capability should be enhanced to allow multiple application ranks to update an index on an object. This would likely be done through an extension to the HDF5 component of the Mercury server that arbitrated access to index updates, although it might be possible to create index storage mechanisms that can be created or updated in parallel.
- A single index query should be able to execute in parallel, leveraging the full bandwidth of the storage system. This may require internal modifications to the index packages themselves (beyond just changes to the plugin that interfaces to them).
- Early feedback from reviewers indicated a desire for storing queries and views within containers, to provide fast access to data that had been generated earlier. Storing queries can leverage the existing capability for serializing them (used by the analysis shipping feature), but additional work will need to be done to store views in a container.
- Additional query targets should be defined and implemented, such as range queries on map keys and values, metadata for datasets (such as array rank or size, or datatype), etc.
- The query execution module in HDF5 needs a formal model, so that it is possible to algorithmically define query optimizations.

- Analysis shipping is a powerful technique for bringing application operations to execute locally on data in containers, but it is reliant on well-written Python code to leverage the capability. A library of Python routines for analysis operations should be developed and provided to application developers. Ideally, it would be possible to embed these routines in containers for future use, or even automatic invocation when triggered by user-defined operations.

7.4 ACG

While the focus was mostly on in-core Graph Analytics, investigating more on out-of-core graph computing would be a natural extension to many interesting questions. The transactional semantics, as outlined earlier, would be a great vehicle for supporting out-of-core graph-parallel computing.

One lesson from parsing many real life data sets, is that it is often very hard to limit everything to fixed length data representation; this requirement often introduces additional steps in the workflows. Further investigation is recommended to efficiently represent variable-length structures, and then take advantage of such structures·

Finally further work should be done to take more advantage of analysis shipping. In big-data applications, very often small analyses are performed on a small subset of data and then small pieces are aggregated for a large-scale overall computation. A really interesting approach here would be to push the small analyses close to the data, and only pull the required pieces to the compute nodes.

7.5 Other top-level APIs

The EFF stack was designed to support multiple top-level APIs that leverage common underlying components and further work would be welcomed on as broad a front as is reasonable. A simple API that exposes the various application objects implemented by IOD but is geared towards application developers rather than middleware developers would be

desirable. Alternatives to HDF5 such as ADIOS, the Adaptive I/O System from ORNL and Georgia Tech. and SciDB, the Open Source Scientific Data Management System should also be considered to validate the architecture and provide comparison and contrast. Other domain specific languages wishing to perform efficient IO in the exascale era will need help to evaluate which layer in the EFF stack is most appropriate for their needs.

References

The following program page is the repository for current versions of the documents referenced below unless otherwise noted:
https://wiki.hpdd.intel.com/display/PUB/Fast+Forward+Storage+and+IO+Program+Documents

ACG Project Documents and Presentations

[A1] ACG Solution Architecture

[A2] ACG Design Document

DAOS Project Documents and Presentations

[D1] Design Document for Reduction Network Discovery

[D2] Design Document for Reduction Network - Notifications and Collectives

[D3] VOSD Design Document

[D4] DAOS API and DAOS POSIX Design Document

[D5] Epoch Recovery Design Document

[D6] Lustre Restructuring and Protocol Changes Design Document

HDF5 Project Documents and Presentations

[H1] The *Design and Implementation of FastForward Features in HDF5*

[H2] *User's Guide to FastForward Features in HDF5*

[H3] *HDF5 Data in IOD Containers Layout Specification*

[H4] *High Level Design - Function Shipper*

[H5] *Burst Buffer Space Management - Prototype and Production*

[H6] *Deep Dive - Transactions* - Powerpoint presentation given September 2013

[H7] *Mercury Design Document*

[H8] *AXE Design Document*

IOD Project Documents and Presentations

[I1] IOD Solution Architecture

[I2] IOD API

[I3] IOD Design Document

[I4] IOD KV Design Document

[I5] IOD Object Storage on DAOS Document

Cross-Company Documents and Presentations

[EFF1] *End-to-End Data Integrity in the Intel/EMC/HDF Exascale IO* - Powerpoint presentation given September 2013