Date: July 2, 2014	# IOD Design Document **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O**

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

1 Table of Contents

Table of Figures

2 Introduction

I/O dispatcher (IOD) is software that runs on the hardware I/O nodes (IONs) which are equipped with persistent solid-state burst buffers. Together with innovations in DAOS and HDF, IOD will change both the hardware storage tiering and software I/O stack to satisfy both the scalability and performance requirements for extreme scale HPC storage system. IOD absorbs application's I/O and buffers it on local SSD.

It provides storage for user data in structured array objects and unstructured "blob" objects as well as providing a first class key-value object. Upper layers can use these objects however they like, but we expect for our project that HDF will use array objects to store users' HDF Datasets and blob objects for unstructured data and KV objects for internal HDF metadata linking the objects within an H5File.

IOD further persists/pre-fetches the data to/from central storage (DAOS) via explicit burst buffer management exported to the upper layer. IOD handles the impedance mismatch between the smooth streaming I/O required for efficient backend disk utilization and the bursty, fragmented and misaligned I/O that frontend extreme scale applications will produce as well as providing mechanisms for efficient analysis and reading of in-transit data as it passes through the IONs. Additionally, it can be used for out-of-core analysis as data sent to ION can be ephemeral and never stored persistently on DAOS.

Four main characteristics of IOD are: object storage, transactions, semantic awareness, and asynchronous operations.

- **Object storage**. IOD discards traditional POSIX semantics and maps complex science data models to container and objects, provides direct access to underlying storage objects to avoid lock contention, allows applications to choose the degree of parallelism related to access needs by providing optional control over where and how objects are striped on underlying storage targets. IOD exports three types of objects: multi-dimensional arrays, key-value stores, and *blobs,* which are akin to POSIX files with some exascale features.

- **Transactions**. IOD provides transactions which ensures a group of operations executed across an arbitrary set of processes within a single parallel job across a set of objects within a single container are applied atomically – i.e. all or none will succeed. It can be used to guarantee the integrity and isolation of the stored science data models.

- **Semantic awareness**. IOD can understand the dimensionality of multi-dimensional data structures based on which it can do layout resharding according to users' requests to allow collections of sub-objects to be stored together on a single ION to enable analysis tasks which require that collection to be read entirely from the local ION. IOD leverages the fast network interconnect between the IONs and can do MPI communications between them for data shuffling. IOD provides APIs to control burst buffer's pre-fetch from or "persist" to central backend storage with optimized layout using semantic descriptions of array dimensions. [Similar functionality is for non-structured objects (called IOD blobs) but this is called "multi-format replicas," whereas we call it "semantic resharding" for structured "array" objects and structured KV objects.]

- **Asynchronous operations**. IOD API is fully asynchronous to allow user can build fully non-blocking applications through which further improves parallelism by overlapping computing and I/O. One IOD API's success return just means the request has been submitted to IOD, a related completion event can be polled by user to query completion. Note that asynchrony is not magic; asynchronous reads and writes on data buffers do NOT immediately allow the application to user those buffers. The read buffer must be protected until the read event completes and the data in the write buffer is not valid until the write event completes. The application must have other work to do in order to benefit from asynchrony. If the application needs the buffer, then the asynchrony doesn't benefit them. Therefore, asynchrony is only one part of the solution. We need fast burst buffers to reduce the time to event completion and we need asynchrony to allow the user to do other work while they wait for the completion.

3 Motivation

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.

The petascale era, although it was able to continue using much the same hardware stack as the terascale era, also had its own other challenges. In fact the petascale era was extremely challenged by high degrees of concurrent access to shared files due to the *false sharing* problem in which application modifications to distinct file regions intersect within the same regions of the file as shared by the storage system. This sharing is called false because, although the storage system treats these modifications as if there is concurrent access to the same data, in fact there is not. To address this, PLFS was developed to decouple concurrent access to shared files to avoid false sharing by sending all data from each process to a separate file. PLFS then reconciles the shared file at read time. The benefits of PLFS are illustrated in Figure 1.

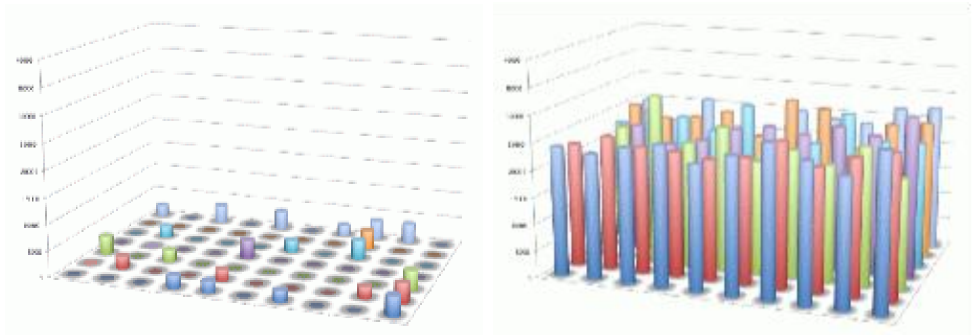


Figure 1. The same shared write workload run on Panasas on the left and with PLFS on the right. Each cylinder is a disk and the height of each cylinder is its current bandwidth. Note that false sharing in Panasas grossly limits total throughput. Image courtesy of Ben McClelland.

This project anticipates new and 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. Even today, the Titan supercomputer, operating on data volumes orders of magnitude smaller than exascale, is experiencing almost daily incidents [Cornell Wright, LANL]. 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.

4 Definitions

IOD

I/O dispatcher software

IOD Daemon

In this document, we often discuss IOD daemon processes which run on each ION, are connected to each other with an MPI communicator, and to the application through the HDF VOL client/server. To be precise, although they function like daemons and we often refer to them as such, they are actually just running as a library linked to the VOL servers.

CN

Compute node

ION

I/O node

BB

Burst buffer

DAOS

Distributed Application Object Storage[4]

User

We use user in this document to refer to whatever is the higher-layer that calls into IOD. Typically this will be HDF although it is probably the application that is making many of the decisions and relaying those decisions to HDF which then in turn relays them to IOD.

Shard

There is a "shard" concept at both IOD layer and DAOS layer. At DAOS layer it means the virtual storage target of DAOS container, similar with current Lustre's OST. At IOD layer it stands for the split data pieces across multiple storage devices comprising an IOD object.

PLFS

Parallel Log-structured File System[2]

Function shipper

An I/O forwarding layer that ships function calls from CN to ION[6]. It is client-server model that client runs on CN and server on ION.

Process group

The "process group" in this document stands for client side application's process group. It is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1.

IOD Container

- This is similar to a POSIX directory. HDF uses an IOD container to store an H5File and an IOD container has a 1:1 mapping to a DAOS container.
- Can contain any number of objects inside which stores user's data.
- In our initial implementation of burst buffers on ION's, we may export the burst buffers through POSIX with a local file system such as ext4. In this case one IOD container will correspond to one "special" directory at every ION. The directory path is IOD container's path and can be visible by POSIX namespace and is how PLFS containers for shared files (N-1) work. This will be the initial implementation of the prototype which will be demo'd. However, we recognize that the number of objects within a container may be very large and this could incur excessive overhead to create a directory for each. Therefore, for future potential productization, we are also working on two alternate designs for this:
    - using PLFS small-file mode which would reduce the number of physical entries as well as not requiring a directory for each logical object
    - using DAOS as the interface to the flash and exporting each flash device as a DAOS shard

IOD Object

- HDF will typical store its objects in a 1:1 mapping with IOD objects and perhaps use extra IOD KV objects to store metadata about these objects. However, DAOS objects are not themselves striped across multiple DAOS storage targets. Therefore to provide parallelism for large objects, an IOD object can be sharded across multiple DAOS objects.
- Three types: array, blob and KV.
    - Array objects are for storing structured multi-dimensional data structures.
    - Blob objects are analogous to POSIX files: simple 1D arrays of bytes.

- KV is a parallel KV store implemented with a modified LANL MDHIM.

5 Changes from Solution Architecture

Only one change from IOD SA document: object versioning. Transactions on the IONs function as temporary versions and can therefore be used for time series analysis but these transactions do not become DAOS versions. If the user wants to *persistently* store multiple versions, then they must ask IOD to persist multiple transactions from ION to DAOS and then use the DAOS "snapshot" mechanism to create new containers instead of having multiple versions of the same container. Importantly, however, DAOS does do efficient copy-on-write to implement "snapshots."

6 Specification

6.1 High level system view

High-level system view is depicted as Figure 2. [The server-side VOL plugin which is provided by HDF to receive data sent by the client VOL on the CN is not shown on the diagram.] This diagram shows the possibility of multiple process groups sharing ION's but this is not our expected workload; our expected workload is a large parallel application on the CNs and perhaps a secondary analysis program running directly on the IONs. However, to share ION data, the application on the CNs and the analysis program on the IONs must somehow be connected to the same set of IOD processes.

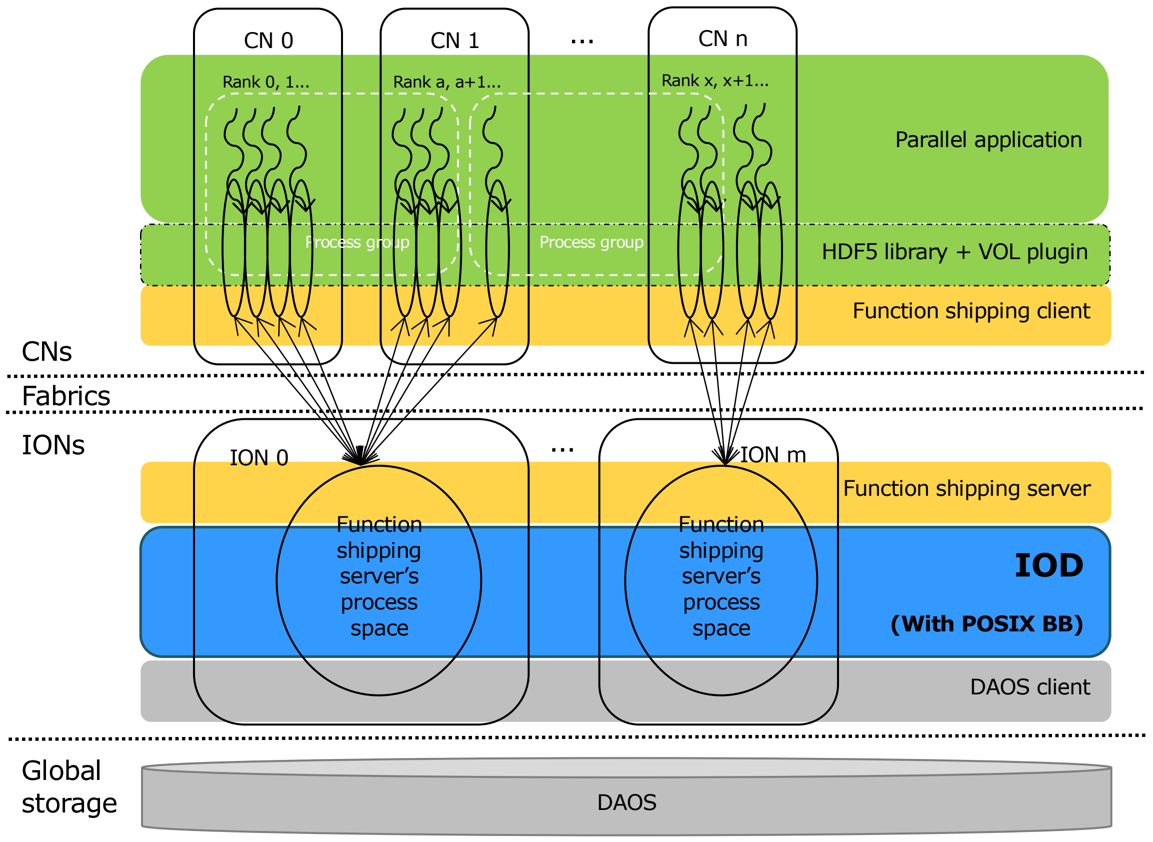


Figure 2. High level system view.

IOD is a library which provides I/O services to upper-layers. IOD doesn't have its own process space; instead it is linked into application's process space. In the case there is function shipper between CNs and IONs so the IOD is linked into function shipping server's process space (the HDF VOL server). IOD will create some service threads within caller's process space. Every ellipse in Figure 2 corresponds to one process which has an independent process space.

The function shipper forwards VOL function calls from CNs to IONs. The function shipper server is a MPI program runs over IONs cluster, it calls IOD's initialization routine and passes in the MPI communicator. Therefore, each IOD process is bound within an MPI group to its other IOD siblings. The IOD siblings can use unexpected MPI messages to coordinate and scatter-gather data as necessary.

This forwarding architecture offloads I/O functionalities from CN to ION, improves system's performance and scalability, but does add some extra complexity to both function shipper and IOD:

- Function shipper only forwards CN ranks' I/O call 1:1 to ION, and one function shipping server provides service to many CN ranks. This may cause IOD to receive some duplicate function calls. For example all CN ranks call IOD object open, one function shipper server possibly will open the same object multiple times, same for object close. So IOD will need to maintain an open ref-count for safe open/ close. The object create is more tricky; to avoid possible race, IOD restricts the object create can only be called once by one CN rank, that rank can get back an object ID and can share it to other ranks for further open.
- Because the function shipper is between a set of user processes and a single IOD instance, IOD may not have information about process groups. Therefore, it knows when transactions are finished by reference counting. Transactions are fully asynchronous but each process that participates must know the number of other processes that participate in that transaction. Each will end independently and IOD will know when the transaction is finished when its reference count drops to zero. An application can also optionally appoint a leader to start and end a shared transaction; this reduces cross ION communication between IOD siblings but prevents the upper layer from committing fully asynchronous transactions.
- Completion of asynchronous events is done with an event "handle" that is not global and only can be queried within the address space of the original caller of the asynchronous function.

6.2 IOD sub-modules overview

Figure 3 shows the high-level overview of IOD's sub-modules.

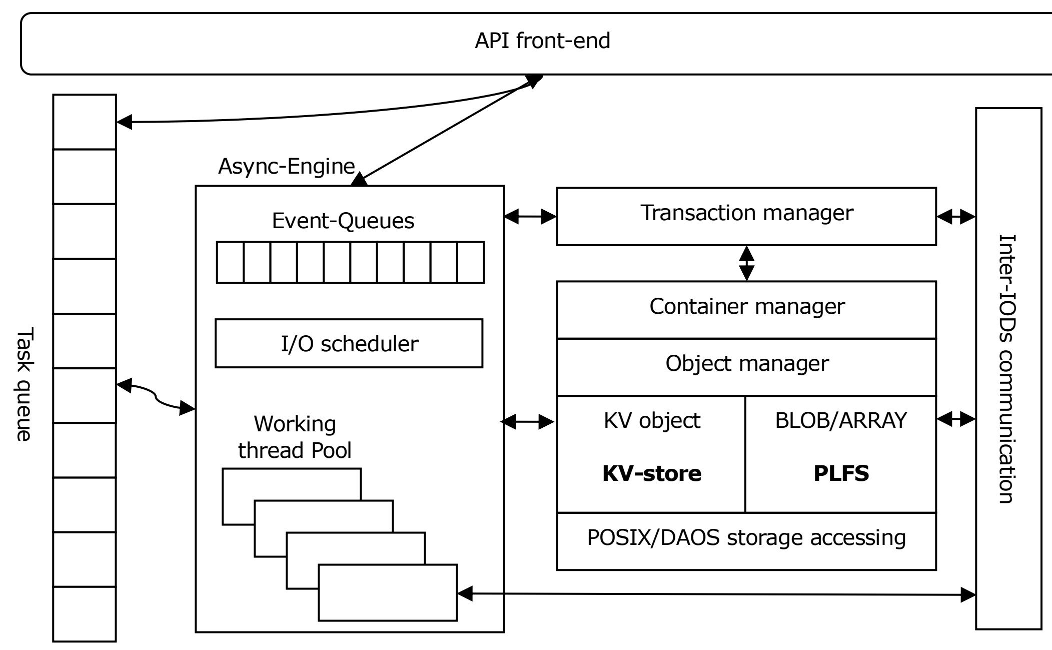


Figure 3. IOD sub-modules overview.

When an IOD function is called by upper layer, IOD inserts an appropriate task to its internal task queue and bonds it to an asynchronous completion event. The task queue is not exactly one single queue or linked list, instead it is a set of task lists which may belong to container or objects, and bind to transactions.

Async-Engine. All tasks are executed by async-engine which is the center for executing and progressing all asynchronous operations. Three main components for the async-engine are thread pool, I/O scheduler, and events and the event queue manager. The thread pool is the executing unit of async-operations, it needs to be initialized inside iod_initialize() with a configurable value for the number of threads. The I/O scheduler progresses asynchronous operations, it also has chance to do I/O optimizations such as stream transformation/merging. The event and event queue manager maintains the relationship between event and pending tasks.

Transaction manager. The transaction manager provides transaction semantics. One specific transaction's status is managed by the transaction manager which is selected by hashing transaction ID. All transaction's final status is tracked by container manager which is selected by hashing container path name.

One IOD container, in the initial implementation, will correspond to one POSIX directory on every ION, and has a backend DAOS container for data persistence, central storage, and write locking. It can have many objects inside one container. IOD's local object storage is based on KV-store and PLFS. IOD have a separate DAOS interface for persisting data and also performing some IOD container operations (for example open, create, unlink etc.) to DAOS. Containers will be synchronously created on DAOS but objects and data will be buffered on IONs. Therefore ephemeral data that is never persisted to DAOS is possible. In these cases, the containers on DAOS will be removed of course.

The **inter-IODs communication** layer is another important module used for communications between IODs. The communications include container/transaction status query/synchronization, possible data shuffling/movement, and internal collective communications etc. There are possibilities that IODs need to do internal collective communication such as for global PLFS index building, object creating etc. IOD can build an internal spanning tree based on which to do the asynchronous collective communications; this optimization is a potential roadmap addition in future quarters. IOD will have special threads listening on unexpected MPI messages from siblings with some pre-defined messaging formats. The communication is based on MPI mechanism, so IOD's caller (function shipping server in our scenario) should be a MPI program and should create the MPI communicator of all IODs. However the function shipping server needs only create the MPI communicator and pass it to IOD when calling iod_initialize(), no other IOD functions take an MPI communicator as a parameter. Examples of communications are for reference counting in which one IOD is elected, based on a hash, to be transaction leader and does the reference counting on that transaction. Also, cross node communication is used for data movement in a scenario where the user has requested to persist a container to DAOS and each IOD may collect small chunks of data from its siblings so that it can send large chunks of data to DAOS.

Finally, IOD has a debug/diagnose supporting layer which is not depicted at above diagram.

6.3 Object storage

Three object types – KV, ARRAY and BLOB

IOD provides three types of object abstractions.

- KV[3]

  It is used to store key-value pairs, such as HDF5 group/attribute/link etc. IOD provides KV store based on MDHIM (Multi-Dimensional Hierarchical Indexing Middleware). IOD exports the KV-store to upper layer through a set of KV-APIs to allow caller to create/open/set/get/list/close/unlink the KV object's content. IOD may also use its own KV stores for some of its own internal metadata such as the object list within a container, and the mapping between IOD object and DAOS objects etc.

- Array

  Array is a multi-dimensional data array which can be used to semantically store an HDF5 dataset. The array object has spatial structure; by understanding the dimensionality, IOD can do a "semantic resharding" where a user requests, using dimensional descriptions, a reorganization of their data (e.g. stripe the array along the vertical faces of the cubes). IOD array object can be extendable along a single dimension, more precisely it can be extended only along the first dimension as specified in its creation. This constraint is important to allow IOD to make layout decisions that won't require reorganization when the object grows. IOD supports similar concepts as HDF5 dataset's contiguous layout and chunked layout[8] to make it smooth to bridge to HDF5 users' common usage.

- Blob

A data blob is simply an stream of bytes and is semantically identical to a standard POSIX file with the addition of transactional semantics

In the current implementation, both the array and blob objects are stored using a modified version of PLFS. IOD uses a PLFS logical file to store the array and blob objects; on IONs, one PLFS logical file is implemented using a PLFS container which is implemented with a set of POSIX directories across IONs. In ION's local storage, one IOD container is a POSIX directory, every array or blob object is a sub-directory inside the parent container's directory. The array or blob's data is stored in PLFS data log files. By leveraging the log-structure of PLFS, IOD provides fast writing speed based on local SSD. User can persist object's data to DAOS, for the persisting IOD will consult the PLFS indices to construct the consistent view of the object and do an optimized layout placement to DAOS. Essentially the array and blob objects will be stored as logs on IONs and flattened striped on DAOS.

Object mapping between HDF5/IOD/DAOS layers

Table 1 shows a possible mapping between high-level HDF5 objects and IOD abstractions. One key thing to notice is that some abstractions do not exist at all layers. For example, an H5Dataset is an IOD array object but there is no such DAOS abstraction; instead, IOD stores an IOD array object as shards across a set of DAOS objects. Since each DAOS object is stored on only one shard then parallel IO to an IOD object requires that the IOD object is stored across a set of DAOS objects.

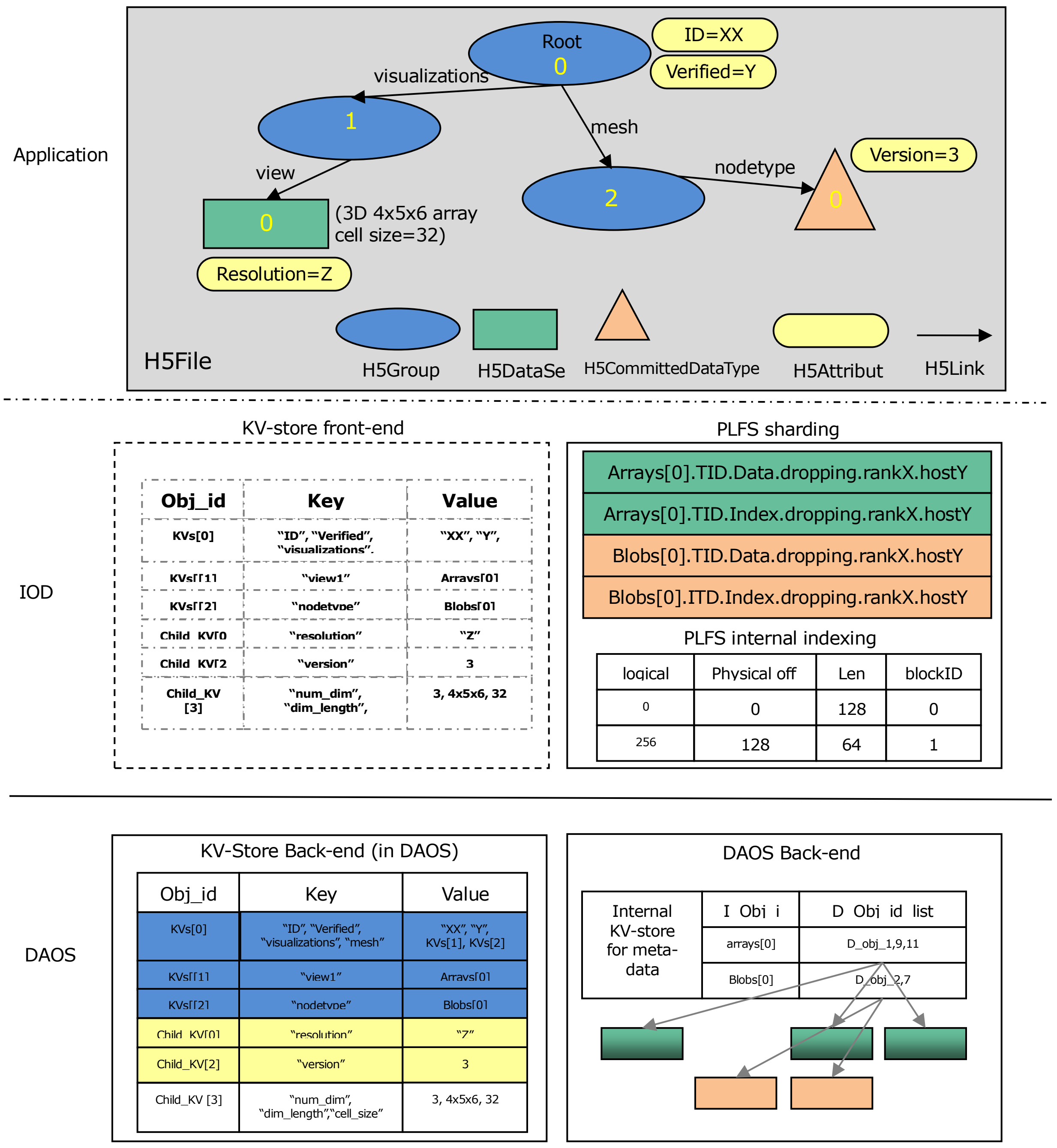
HDF5 Abstraction	**IOD Abstraction**	**DAOS Abstraction**
H5File	Container	Container
H5Group	KV object	Set of DAOS objects
H5DataType H5DataSpaces H5Attribute H5Properties H5Reference H5Link	KV pair in KV object	Data in a DAOS object
H5Dataset	Array object	Set of DAOS objects
H5CommittedDatatype	Blob object	Set of DAOS objects

Table 1. Object mappings across layers. One caveat is that H5Dataset will use a blob however if the elements aren't fixed size. Also, HDF will create additional KV objects for storing metadata about some of the objects and their additional attributes.

Figure 4 gives out a more detailed example to illustrate the object mapping at different layers. The example includes HDF5 group/link/dataset/committed-data-type/attribute objects and how they are mapped and stored on both IOD and DAOS (after migration). Many more details are available in the respective design documents for HDF and DAOS.

In Figure 4, there are 3 H5Group objects which are implemented by IOD KV objects, one H5Dataset and H5CommittedDatatype objects which can be implemented by IOD array object and blob object respectively. For the root group, the 2 H5Attribute objects "ID=XX, verified=y" can be implemented by KV-pairs. The first created IOD KV object is root group. After creating the root group, which is always assigned IOD object ID 1, user can create other kinds of objects and use KV-pairs to establish the relationship between them, for example the "visualizations" link points to root group's sub-group and a further link "view1" points to the H5Dataset object.

User can create similar "child-parent" relationship between some objects, for example to store the H5Attribute objects belong to H5Dataset or H5CommittedDatatype object ("resolution=z" and "version=3" in the example). Note that the child-parent relationship (the child KV objects on Figure 4) can be implemented by the "scratchpad" of parent object. IOD provides fixed length (32 bytes for example) storage as a scratchpad which can be associated with any type of IOD object. User can store the child object ID inside that scratchpad. IOD provides interface to get/set the scratch. This is similar to POSIX extended attributes. IOD will internally store all objects' scratchpad using one special KV object. Because the IOD KV objects are, of course, also transactional, these scratchpads will be as well. The "ID=XX" and "Verified=Y" attributes in figure 3 possibly also will be implemented by a separate KV object and store the object ID in root group's scratchpad.



Obj_id	Key	Value
KVs[0]	"ID", "Verified", "visualizations",	"XX", "Y",
KVs[[1]	"view1"	Arrays[0]
KVs[[2]	"nodetype"	Blobs[0]
Child_KV[0	"resolution"	"Z"
Child_KV[2	"version"	3
Child_KV [3]	"num_dim", "dim_length",	3, 4x5x6, 32

Arrays[0].TID.Data.dropping.rankX.hostY
Arrays[0].TID.Index.dropping.rankX.hostY
Blobs[0].TID.Data.dropping.rankX.hostY
Blobs[0].ITD.Index.dropping.rankX.hostY

logical	Physical off	Len	blockID
0	0	128	0
256	128	64	1

Obj_id	Key	Value
KVs[0]	"ID", "Verified", "visualizations", "mesh"	"XX", "Y", KVs[1], KVs[2]
KVs[[1]	"view1"	Arrays[0]
KVs[[2]	"nodetype"	Blobs[0]
Child_KV[0]	"resolution"	"Z"
Child_KV[2]	"version"	3
Child_KV [3]	"num_dim", "dim_length","cell_size"	3, 4x5x6, 32

Internal KV-store for meta-data	I Obj i	D Obj id list
	arrays[0]	D_obj_1,9,11
	Blobs[0]	D_obj_2,7

Figure 4. An example object mapping. Approximate only; some simplifications made to facilitate the transmission of the general idea.

For array's data space information such as dimensions, length, cell size etc, IOD will create a corresponding internal KV object ("3D 4x5x6, cell_size=32" as above example) and associate it with the array object. The array or blob object's data can be sharded to multiple IONs' local SSD by PLFS logic. After migration IOD will create a set of DAOS objects to store the data into DAOS, and IOD will store its own metadata such as the mapping between IOD object and DAOS object list using an internal KV-store (the "Internal KV-store for metadata" shown on Figure 4) which uses another set of DAOS objects as its backend storage.

Object storage on DAOS

IOD stores and buffers all kinds of objects (KV, blob and array) on local storage of ION, and only when application explicitly calls transaction persistence (through iod_trans_persist) it will read those buffered data out from ION's local storage and write to DAOS global storage. In the inverse direction, user can explicitly call IOD's API (iod_obj_fetch) to fetch objects (or sub-objects) from DAOS to ION.

As is also shown in Table 1, the main abstraction mapping between IOD and DAOS:

- An IOD container maps to a DAOS container
    - A DAOS container is a group of DAOS shards
    - A DAOS shard can host a set of DAOS objects
- An IOD object (KV, blob or array) can be stored across a set of DAOS objects. This is a key point: there is not a 1-1 mapping between IOD object and DAOS object. Each DAOS object is local to one DAOS shard which is local to one DAOS target. In order to achieve parallel storage access for large objects, IOD must necessarily spread data for large objects across multiple DAOS shards.

When writing object to DAOS:

- IOD will create some DAOS objects (one DAOS object for every DAOS shard) for storing that IOD object's data. IOD's object ID is 64 bits length, and DAOS object ID is 128 bits which includes 64 bits DAOS shard ID and 64 bits DAOS object ID within the shard. When IOD writes IOD object to DAOS, it will combine 64 bits IOD object ID with a set of 64 bits DAOS shard IDs to form a set of valid 128 bits DAOS object IDs and store that IOD object's data to those DAOS objects.
- IOD will shard one IOD object to a set of DAOS objects, and can keep the exactly same address space between IOD object to DAOS objects because DAOS object's address space is virtual and unlimited, so within every DAOS object there is possibly many big holes inside its address space – IOD combines all those DAOS objects' address space for one IOD object's address space. This can reduce the needed metadata to establish the mapping between IOD object and DAOS objects.
- For storing on DAOS, IOD will consider DAOS' shard property to select appropriate shards. For example some DAOS shards are optimized for bandwidth, IOD will use these shards to store bulk raw data; and some shards are optimized for IOPS, IOD can use for storing metadata.
- When persisting, IOD will use the same DAOS epoch number as IOD TID to make them 1:1 mapped. But not all IOD TIDs need to be explicitly persisted to DAOS, i.e. some epoch numbers on DAOS possibly will not be used.
- When creating IOD container, IOD will create a corresponding DAOS container and allocate some DAOS shards within the DAOS container. And IOD can dynamically add new DAOS shards to DAOS container when user wants to use more DAOS target storage.

IOD needs to store some metadata to manage its objects. For objects newly written to ION, IOD metadata may be large: the maximum is one metadata entry per write from HDF, althought aggregation or pattern discovery[11] may reduce this somewhat.

The metadata for describing object layout on DAOS is much smaller since IOD reorganizes data onto DAOS in large flattened stripes. IOD will maintain related metadata to store the mapping from IOD object to DAOS objects. IOD can use an internal KV-store to store those internal metadata. This is the metadata maintained by IOD:

- The list of IOD objects within container
- The mapping from each IOD object to DAOS objects
- The layout (sharding and striping) of the IOD object
- The maximum valid offset of the object

Figure 5 shows an example of a blob object being stored on DAOS and Figure 6 shows an example of an array object. Section 4.4.1 will further introduce the data migration and section 4.5.3 will discuss the consistency semantics between IOD and DAOS related to transaction/epoch.

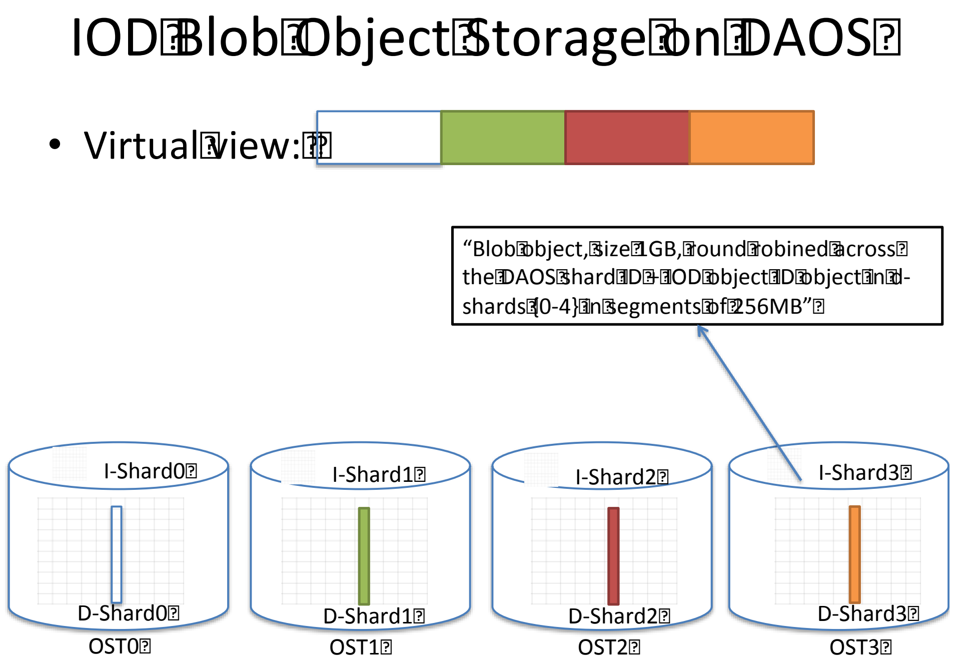

IOD Blob Object Storage on DAOS
• Virtual view:
"Blob object, size 1GB, round robined across the DAOS shard ID - IOD object ID object in d-shards {0-4} in segments of 256MB"
I-Shard0
D-Shard0
OST0
I-Shard1
D-Shard1
OST1
I-Shard2
D-Shard2
OST2
I-Shard3
D-Shard3
OST3

Figure 5. How IOD stores a blob object on DAOS. Note how the object is stored across DAOS shards using regular round-robin striping. This allows two important benefits: first, sets of over-writes to multiple objects are handled atomically by DAOS transactions and DAOS does garbage collection for over-written data, and second, the amount of IOD metadata for locating physical data is reduced. Note that this figure shows the IOD metadata stored in a separate index shard whereas the current implementation creates a virtual index shard and a virtual data shard and stores them together in a single DAOS container shard.

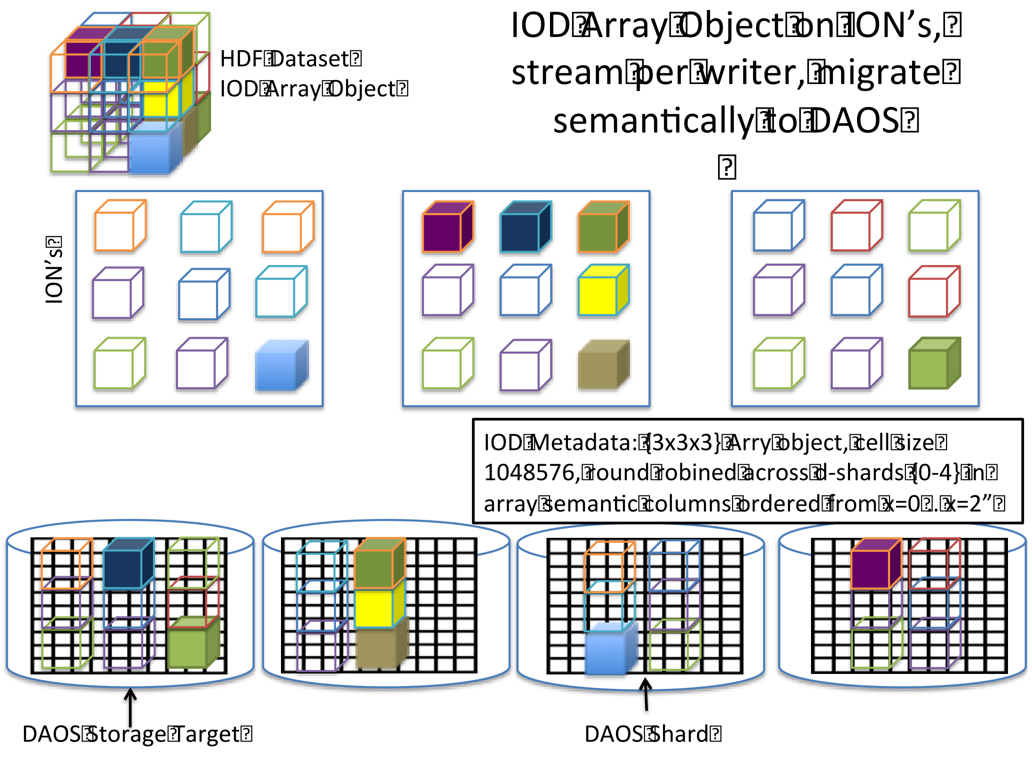

HDF Dataset
IOD Array Object
IOD Array Object on ION's, stream per writer, migrate semantically to DAOS
ION's
IOD Metadata: {3x3x3} Arry object, cell size 1048576, round robined across d-shards {0-4} in array semantic columns ordered from x=0 . x=2"
DAOS Storage Target
DAOS Shard

Figure 6. How IOD stores an Array Object on DAOS. The app can request a striping layout and can query the number of shards if it wants explicit control over parallelism or it can rely on IOD to make a reasonable layout based on the number of shards.

6.4 Storing organization of data, metadata, and checksums

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:

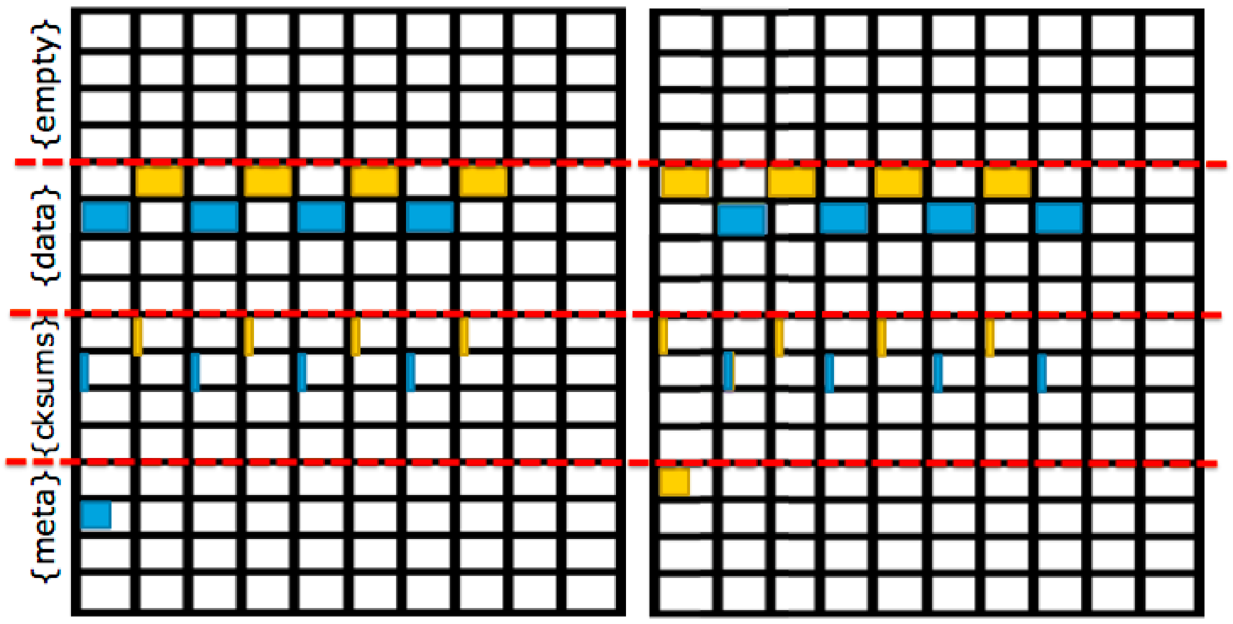


Figure 7: Two IOD Objects Distributed Over DAOS Container Shards. Data and metadata and checksums for the blue IOD object is stored within the third object within their respective virtual shards as the yellow object is stored in the fourth.

Figure 7 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 7 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.

6.5 Layout, data migration and reorganization

One of the most important and complicated aspect of IOD is the data layout, migration and reorganization across the two storage tiers. For this, IOD's has three major goals: a) be semantic/structure aware and provide flexible interfaces to allow callers to control the data layout, sharding granularity and placement across DAOS shards and IONs; b) provide an API well-suited for exascale HDF; c) reduce the metadata needed for logical/physical mapping, sharding placement, etc.

There are mainly two kinds of mappings that IOD needs to maintain:

1) Maps the logical spatial space of array objects to a logical one dimensional address space (like a blob object) – this is handled in the new IOD portion of the code,
2) Maps the logical one dimensional address space (either directly for a blob object or for an "unraveled" array object as above) into a set of physical addresses on different storage devices – this is handled in the modified PLFS portion of the code

6.5.1.1 High-Level Data movement semantics

The IOD is responsible for data movement into and out of both the ION and DAOS storage targets as is shown in Figure 8 and described in Table 2.

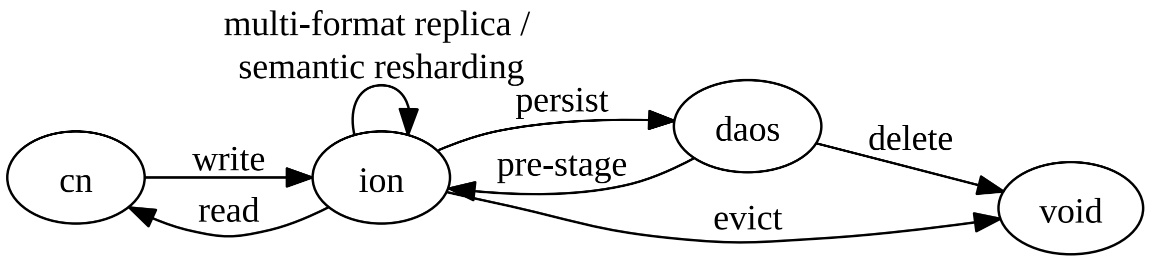


Figure 8. IOD managed data movement. Multi-format replica is for blobs and KVs. Semantic resharding is the same idea but for arrays. Void means that the data is no longer available from that layer. Evict and unlink operate on entire IOD objects.

SRC	DEST	TRIGGER	LAYOUT	ACTION	GRANULARITY
CN	ION	write	Logged	The data received will be appended to a PLFS style data log in the burst buffer and a PLFS style index entry will record it.	Sub-object
ION	CN	read	n/a	The ION receiving the read request will fetch any needed data from its siblings or DAOS if necessary.	Sub-object
CN	DAOS	n/a	n/a	The CN's cannot write directly to DAOS. Their writes will always go first to the ION's. Of course, the design allows for ION's to not be present but this is outside SOW scope.	n/a
DAOS	CN	read	n/a	Happens when requested data is not on any of the IONs. IOD on the IONs will fetch the data into a temporary memory buffer on the ION and then send it along to the CN.	Sub-object
ION	DAOS	persist	Striped	The user can request that the view of the container at transaction t be persisted to DAOS. In this case, the IOD's will coordinate to scatter-gather data so that each DAOS shard has a single writer and the resulting objects on DAOS are striped in a round-robin fashion. Following the migration, transaction t will be the DAOS HCE and will be a consistent view of all objects in the container at t. EXAMPLE. Transactions t and t-1 are on IONs. The user requests a migration of t. IOD will start transaction t on DAOS, and will write all of t and any pieces of t-1 that were not overwritten by t directly into their round-robin stripe locations on DAOS. Then it will commit transaction t to DAOS.	Container
DAOS	ION	pre-stage	Striped, Explicit	At the granularity of sub-objects at a particular transaction or DAOS HCE, the user can request that IOD pull data from DAOS and store it on the IONs. The user can specify irregularly shaped sub-objects and dictate their layout on particular IONs. This data is then readable from IOD. EXAMPLE. The user requests that from DAOS HCE that the first two hyperslabs of array object AO1 are stored with the first gigabyte of blob object BO1 on ION1 and the next three hyperslabs of AO1 and the next 512 megabytes of BO1 are stored on ION2.	Sub-object
ION	ION	semantic resharding , multi-format replicas	Striped, Explicit	Same as Pre-stage except that the expectation is that most of the data will already be in an ION in another layout. Any data not on ION will be fetched as necessary from DAOS.	Sub-object

Table 2. IOD managed data movement. All three object types have similar characteristics. Note that evict/unlink as shown in Figure 8 are not included in this table; their granularity is on entire objects (i.e. not sub-objects).

6.5.1.2 Data Layout Policies

There are three methods that we have identified to layout data into each storage layer:

1. **Explicit**. With this method, the upper layer will inform IOD about exactly where every byte (or semantic chunk) of the object should go. For example, it might say to put the first three elements of an array object onto the first ION and the next five elements onto the second ION.

2. **Logged.** With this method, the data resides in a log-structured array of bytes. For example, the data in a write from the CN to the ION will just be appended into a log on that ION.

3. **Striped.** With this method, each piece of data will go to a deterministic stripe in a round-robin distribution across the storage devices (i.e. either IONs or DAOS shards). The user can specify the stripe size which is bytes for a blob object, elements for an array object, and number of keys for a KV object. For array objects, the user can also specify the ordering across the dimensions. For example, in a 3D array, the first element is {0,0,0}. The second element could be {1,0,0}, {0,1,0}, or {0,0,1} depending on the specified ordering.

For the purposes of our demonstrations, please refer to Table 2 to see which layouts will be within our design and Table 3 to see whether and when we will demo each of these layouts for each of our object types. The three methods have different implications in terms of the amount of metadata that IOD needs in order to locate data; explicit and logged will have arbitrary amounts of metadata with one piece of metadata for every sequential range of data. To understand this, this is the same as the index metadata in PLFS today. The striped method will have a single piece of metadata listing which targets, the stripe size, and the dimensional ordering for array objects. Note that logged and explicit are very similar and in fact will have identical styles of logged data and large amounts of metadata; effectively they are the converses of each other. Logged is creating a rich metadata to reflect a data placement imposed by logging writes whereas explicit is created data logs to reflect a rich metadata provided by the user.

One item of particular importance is to notice that we are only supporting a striped layout on DAOS. This is because DAOS is the layer responsible for flattening overwrites and garbage collection of stale data across multiple transactions. If we provided logged or explicit layouts on DAOS, then these services would need to be reimplemented within IOD. [In fact, we *will* have to explicitly do flattening for KV stores by replaying a transaction log of inserts/unlinks etc to each KV store for each transaction.]

	Movement	Explicit	Logged	Striped
Blob	Write	oos	Q5	oos
	Migrate	oos	oos	Q5
	Pre-stage	TBD	oos	Q5
	MFR	TBD	oos	Q5
	SR	n/a (SR for Arrays only)		
KV	Write	oos	Q5	oos
	Migrate	oos	oos	Q5
	Pre-stage	TBD	oos	Q5
	MFR	TBD	oos	Q8
	SR	n/a (SR for Arrays only)		
Array	Write	oos	Q5	oos
	Migrate	oos	oos	Q7
	Pre-stage	TBD	oos	Q7
	MFR	n/a (n/a MFR for KV and blobs)		
	SR	TBD	oos	Q8

Table 3. Whether and when each layout policy for each data movement for each object will be demo'd. LEGEND: MFR: Multi-format replica. SR: Semantic resharding. oos: Out of scope. TBD: To be determined.

6.5.1.3 One dimension extendable data array

In FastForward project, we decided to support HDF5 multiple dimensional dataset with at most one extendable dimension, i.e. the dataset has either all fixed dimension lengths or can be extended in one dimension.

To support this requirement, IOD's data array object is one dimension extendable multi-dimensional array. To allow IOD to calculate a determined address space for that data array, IOD restricts the array object to be extended only along the dimension specified first in the object creation and all other dimensions are with fixed length. It is like HDF5 original "external dataset" except:

- The HDF5 original external dataset's content must be stored by external files. IOD will store the data in array object within the container.
- The HDF5 original external dataset only can use contiguous layout with fixed dimension layout sequence, whereas IOD provides both contiguous layout with changeable dimension layout sequence and chunked layout supporting. Details in next sub-section.

6.5.1.4 Array's layout, migration and resharding

6.5.1.4.1 Layout mapping

The original HDF5 dataset have two kinds of layout – contiguous layout and chunked layout[9]. The below illustrates HDF5 original semantics of layout and IOD's extensions/changes to it:

- contiguous layout
    - The HDF5 original contiguous layout simply flattens the dataset in a way similar to how arrays are stored in memory, serializing the entire dataset into a monolithic block on disk which maps directly to a memory buffer the size of the dataset. And the dimension layout sequence is fixed; using C's row-major order, the first dimension is the slowest changing dimension and the higher dimensions are faster changing, the last dimension is the fastest changing on disk. For example a three dimensional dataset A, then the first element on disk would be A[0][0][0], the second A[0][0][1], the third A[0][0][2], and so on. If the application read by the same layout sequence such as from A[0][0][8] to A[0][0][520] the underneath will be well behaved as sequential read from disk. But in the case if application wants to read from A[8][0][0] to A[520][0][0] it will have worse performance because the under layer will do either lots of random reads or read back large un-needed blocks and sieving the wanted items.
    - IOD supports contiguous layout with extended capability that allows user can change the dimension sequence of layout. For a 3-dimensional data array with X-Y-Z axes, user can control how to flatten those axes on object space – either X-Y-Z sequence or Z-Y-X sequence etc. Same as above example, if user want to read from A[8][0][0] to A[520][0][0] it can set the logical first dimension as the physical last dimension for storing to disk. This is very useful when the simulation program and analysis program have different accessing pattern. The simulation program may write dataset by X-Y-Z axes to disk, later the analysis program wants to read it by Y-Z-X axes then it can pre-fetch the dataset from DAOS and change the layout to Y-Z-X axes and shard it to a set of IONs with it preferred way. We call this semantic resharding.
- Chunked layout
    - HDF5 original chunked datasets are split into multiple chunks which are all stored separately in the file. The chunks can be stored in any order and any position within the HDF5 file. Chunks can then be read and written individually, improving performance when operating on a subset of the

dataset. HDF5's chunk filter, chunk cache etc are applied on chunk granularity. HDF5 uses quite complicate and sophisticated logic of B-tree or skip-list[8] to map chunk indices to file offsets for a chunked dataset.

- IOD's array object supports chunked layout to satisfy HDF5 users' traditional usage, and HDF5 original chunk filter and similar functionality can be smoothly applied on IOD array object with chunked layout. As both IOD layer's PLFS and DAOS' object provide virtual unlimited address space, so IOD can implement the chunk layout and avoid the most complicate things of HDF5 original B-tree or skip-list chunk indexing mechanism. Everything can be calculated by IOD so IOD needs not maintain the indexing to map the chunk indices to file offsets. The chunked layout can only be set when creating the dataset and cannot be changed after create same as original HDF5. The chunk selection is a parameter can be passed in for IOD array object create.

As mentioned above, the layout maps the logical spatial space of a multi-dimensional dataset to physical storage object's one-dimensional address space. Figure 9 gives an example to show the IOD array object's contiguous layout and chunked layout mapping.

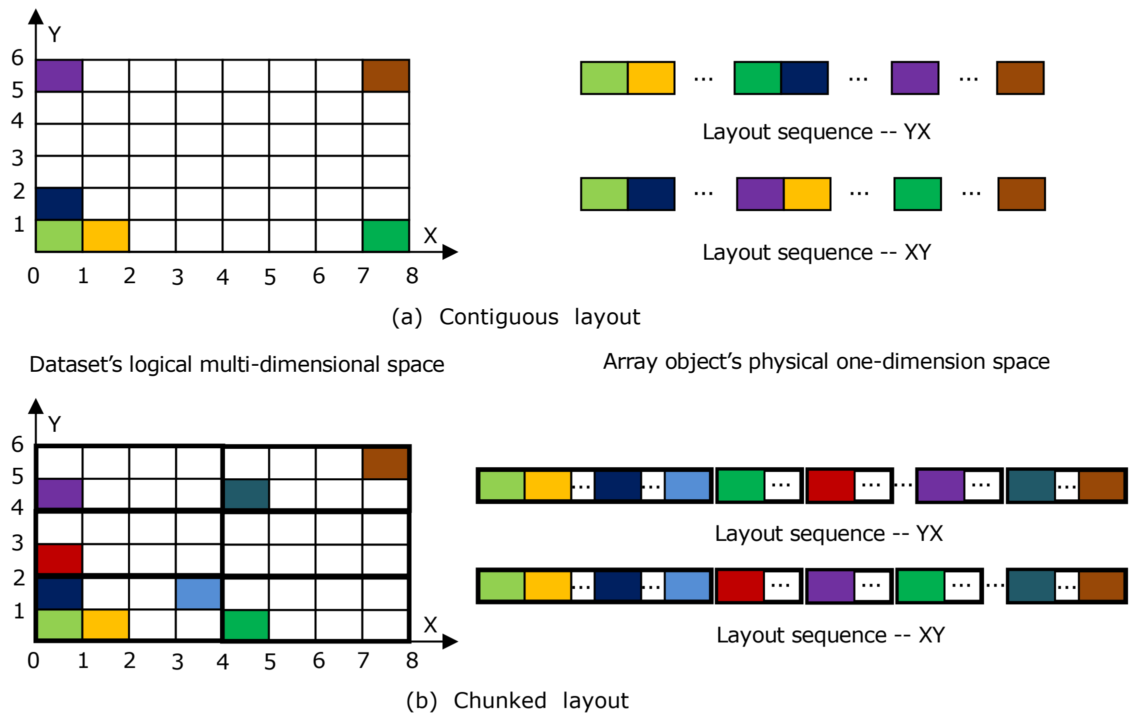


Figure 9. An example of array's contiguous layout and chunked layout.

The example is a 2D 6x8 array. Part (a) shows the contiguous layout mapping, and IOD allows users to change the layout mapping for example from Y-X sequence to X-Y sequence which causes IOD to rearrange the physical layout of data. The dataset's logical structure is not changed – it is a 2D 6x8 array regardless of the physical mapping. The layout just determines the mapping from logical space to IOD array object's storage space.

Part (b) shows the chunked layout that sets 2x4 chunk size, so the array will have 6 chunks in total. For chunked layout, IOD will do write/read with the granularity of chunk. The array's original dimensions will be chunked to smaller chunked dimensions, for the above example the chunked dimension is 2D 3x2. User can further select the layout sequence of the chunked dimension for example Y-X or X-Y. But within each chunk, IOD will not change the physical layout – it is always same as logical dimension sequence, Y-X in the above example. A note here is sometimes the number of dimensions will be changed after setting the chunked layout, for example in the above example the 2D 6x8 array will became 1D chunked array with only 2 chunks if setting chunk size as 6x4, so user needs not set the chunked dimension sequence in this case.

For the extendable multiple dimensional array which can be extended along the first dimension, user cannot change the first dimension's sequence, the extendable first dimension is always the slowest changing dimension, i.e. the logical extendable first dimension must also be the first physical dimension for both contiguous and chunked layout.

IOD will first implement contiguous layout. The chunked layout is only optional and will be implemented when the schedule permits.

6.5.1.4.2 Migration and resharding

By selecting appropriate layout type and dimension sequence, user can control the layout mapping from dataset's logical space to underlying object storage space. Besides this, IOD provides interface to control the data migration and resharding: user can control how to split the flattened object address space into multiple pieces (shards) and place the shards across a set of DAOS shards or IONs. The "sharding/resharding" stands for controlling the shard granularity and placement among storage targets (IONs' local SSD). The sharding granularity is multiple dataset items for contiguous layout, or multiple chunks for chunked layout.

For migration from BB to DAOS, IOD will not automatically free BB's storage space when the migration is done. Similarly, when pre-fetching from DAOS to BB, IOD will not free ("punch") DAOS object. When migration target location is BB, IOD will generate a new TID which user can get for reading the data. The "new TID" is just adding special flags on the original TID; since the TID is 64 bits, IOD will reserve 8 for its own metadata about the TID such as whether it is a replica

The data migration to DAOS is always for entire transaction. User should control the reasonable transaction granularity for it. When moving or replicating data, the below parameters/behavior can be designated by caller:

1) Direction (BB to BB, BB to DAOS, DAOS to BB or DAOS to DAOS), IOD provides different APIs for it.
2) Target layout – the dimension sequence or chunked dimension sequence (X-Y-Z or Z-Y-X for example), if no layout is designated then IOD will use the previously set valid layout, the default layout before any special setting is same physical dimension sequence as logical dimension. Or for movement to DAOS, IOD will use the striped layout.
3) Number of storage targets – DAOS shards or IONs. User can set it as zero in which case IOD will use all available targets.

4) Sharding granularity – how many dataset items for contiguous layout, or how many chunks for chunked layout. All split shards will be round-robin placed on those storage targets as determined by 3). User can set it as zero in which case IOD will select a reasonable granularity (maybe 4M bytes or other value).

User can control kinds of layout and sharding policies, but it should be mainly used for analysis program when pre-fetching from DAOS to BB, or resharding from BB to BB. Once an object is persisted to DAOS, in our initial implementation, the striping parameters will be immutable since the underneath VOSD[10] needs a relative fixed object layout to detect the changed and unchanged extents/ranges. Changing layout on DAOS would cause basically all data to be re-read and then re-written to DAOS and will be outside the scope of our demonstrations.

IOD blob object

The IOD blob object is much simpler compared to array object. Blob size can be grown by appending to it, and user needs not setting/changing the layout as it is only one-dimension. For blob's migration, user only needs to decide the migration direction, number of storage targets and sharding granularity. The blob's sharding granularity is byte as blob is just a bytes stream. Essentially a blob object is an array object with one dimension and one byte chunks.

In quarter 5 of the project, it became clear that ACG would benefit from the ability to have multiple processes atomically append to the same blob object consistently without cross-process coordination. IOD was able to provide this functionality to them. Internally IOD implements this by having the IOD process leader for the object maintain the last offset for the object. Whenever other IOD processes need to append to the object, they query the last offset from the object leader who then increments the last offset by the length of the append. Note that explicit offset writes should not be interspersed into appendable blobs as they may overwrite the appends. In the future, we may improve the scalability of the implementation of appendable blobs by delaying the resolution of the offset for each append until read time. Since IOD stores blobs in the BB's into PLFS files, we will add an atomic append operation into PLFS. PLFS will just mark the offset within its index entry for each append with a special APPEND flag. Then at index resolution time, PLFS will assign an offset. Note that this will allow explicit offset writes to be safely interspersed with atomic appends.

Multi-format replicas

This replication is similar to pre-stage for blobs and kv objects except it should cause ION-ION traffic whereas pre-stage causes DAOS-ION traffic. Same as other migrations, user can set the layout, number of targets, and sharding granularity. The only difference is the replica is a kind of duplicate data in logical concept, but possibly with different physical layout or shards placement. The replication is also in the granularity of transaction.

User can select to do the replication for sub-chunks of objects as they appeared at a particular transaction (assuming that view is still available either on DAOS on from the IONs). The sub-chunks are then replicated into IONs and placed as requested by the user. When reads occur to the multi-format replica for data that isn't in the sub-chunks, then IOD will fetch the required data from DAOS. This means that when the user creates the sub-chunked multi-format replica, that IOD must get a read handle on that DAOS container to ensure that any missing data isn't removed due to flattening while the user still has a read handle on the sub-chunks on the IONs.

Users can also create "bundles" which are collections of sub-chunks which must be located together on the same ION to help analysis routines which need correlated data across a set of objects.

After the success of the replication, the user is returned a new TID (with a special flag transparently set in the IOD reserved bits, which indicates it is a replica). User should explicitly evict the replication's data to free BB's storage space. Multi-format replication on array objects can be referred to as semantic resharding. KV stores may also be semantically resharded by specifying how to partition (and repartition) the key-ranges across the IOD's.

6.6 Transactions

IOD provides transaction semantic to upper layer with the following properties (similar as DAOS[4]):

- **Atomic writes** – either all writes in a transaction are applied or none of them are.
- **Commutative writes** – concurrent writes are effectively applied in TID order, not time order.
- **Consistent reads** – all reads in a transaction may "see" the same version data even in the presence of concurrent writers.
- **Multiple objects** – any number of IOD objects within one container may be written in the same transaction. IOD transaction is at container level.
- **Multiple threads** – any number of threads and/or processes may participate in the same transaction.

Every transaction has an identifier as **TID**. All IOD I/O operations include a TID parameter.

- Read specifies a TID to ensure multiple reads "see" a consistent version of the data. Write (including unlink) specifies a TID to ensure multiple writes are applied atomically. IOD does not allow user to read and write to one TID at the same time.
- IOD TID is not 1:1 mapped to DAOS epoch as some transactions may be only buffered at BB. When a transaction is persisted to DAOS, IOD will use a DAOS epoch number equal to the TID of the transaction.
- TID is a 64 bits value. IOD reserves the highest 8 bits for internal using, for example as replica flags etc.
- For every container, IOD maintains:
    - lowest_durable TID. It is the lowest TID which had been migrated to DAOS and hold a referenced DAOS HCE snapshot.
    - latest_rdable TID. It is the latest (highest) TID which is readable on BB, it possibly has not or has been migrated to DAOS.
    - latest_wrting TID. It is the latest (highest) TID which is started for writing.

Transaction status

A transaction has 7 different possible states as shown in Figure 10:

- **Unborn** – this TID has not been started
- **Started** – not all participants have yet finished
- **Finished** – all participators have finished but one or more earlier transactions are not finished
- **Readable** – all participants have finished and all earlier transactions are also finished or aborted
- **Aborted** – a participator aborted it, any written data will be discarded. For each abort, the user specifies whether it cascades or is independent. Cascading aborts cause all higher transactions (which are necessarily either finished, unborn, or started) to also abort. Independent aborts do not abort higher transactions.
- **Durable** – readable and has been migrated and so is persistent on DAOS.
- **Stale** – previously durable or readable TID for which all of the data is no longer available. These cannot be read but IOD will not automatically evict stale TID. IOD will prevent any transactions which have open read handles from becoming stale.

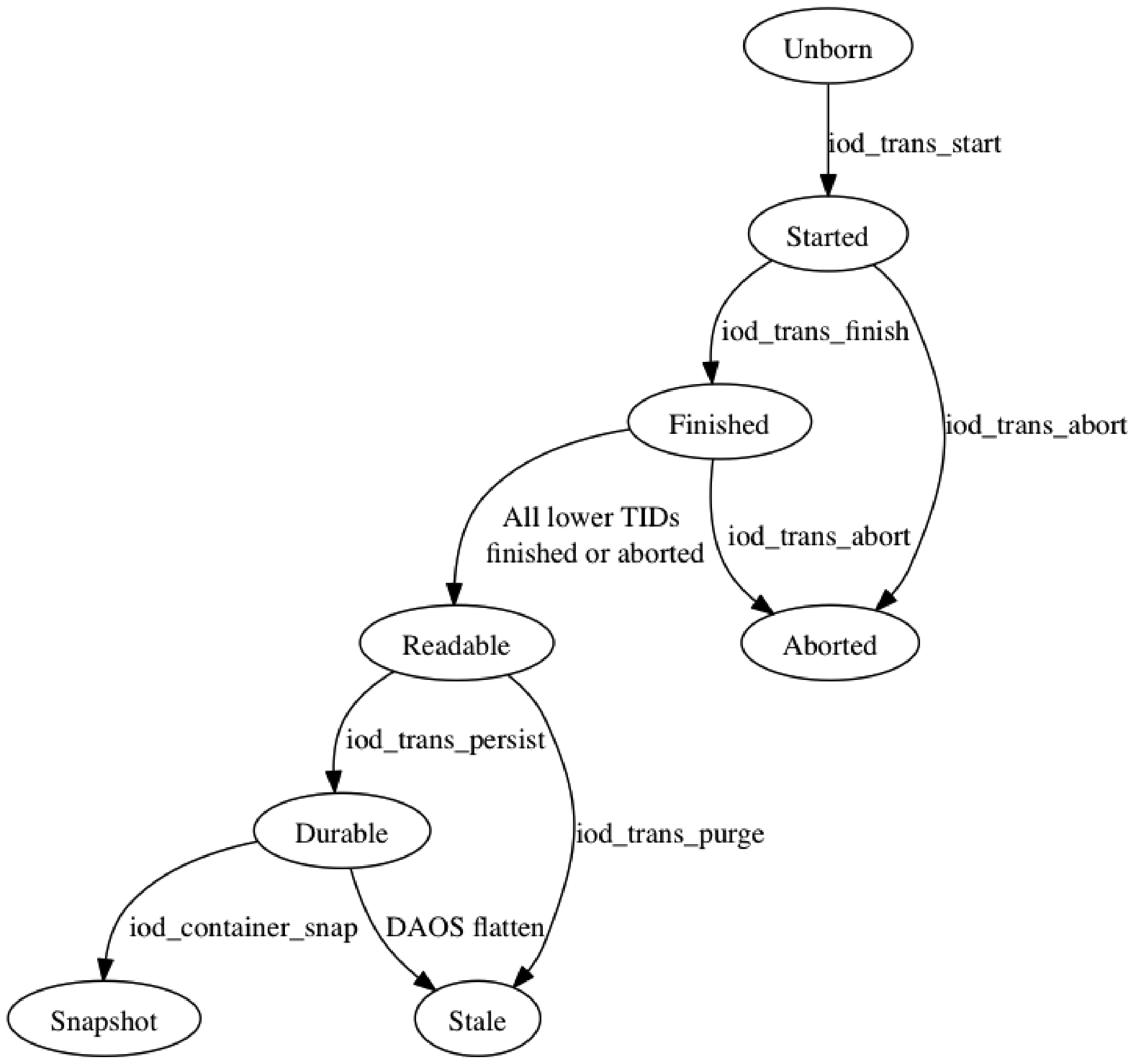


Figure 10. IOD Transaction State Diagram. Note that the user can specify different abort semantics. An aborted transaction can cause higher transactions (which are necessarily either finished, unborn, started, or aborted) to either be unaffected or to be also aborted. Snapshot is included in this figure even though it isn't technically an IOD transaction state. But it is shown since only a container in a durable state can be snapshotted. Snapshotting a durable container does nothing to that container itself; it merely creates a copy of it.

6.7 TID selection

To participate in a transaction, user should first get an appropriate TID by either of the below two methods:

- Can call *iod_container_query_tids* to query this container's TID status.
    - The TID higher than latest_wrting is writable, common use case is to start TID (latest_wrting + 1).

    - For read, the TIDs between lowest_durable and latest_rdable can be readable but possibly there are some unborn or aborted TIDs within the range.
- User can pass in "IOD_TID_UNKNOWN" if it does not want to do the query. IOD will select an appropriate TID and returns to user.
    - For writing, IOD will select (latest_wrting + 1) and return to user.
    - For reading, user can pass in hints to say "I want to read the lowest readable TID" or "I want to read the latest readable TID" as there might be multiple readable TIDs between lowest_durable and latest_rdable TID.

Transaction status synchronization

For every TID, IOD will hash it to one IOD instance as its transaction leader which corresponds to manage/track the transaction status. Besides this, the container manager will track all transactions' final status as well as TID allocation. By this method the transaction leader can partake some workloads from container leader to avoid the container leader being overloaded.

Any number of participators (CN ranks) can participate in transaction, so IOD needs to have an approach to determine the final transaction status among all participators. IOD supports two kinds of transaction status synchronization and consistency ensuring mechanism:

1) Application does the transaction status synchronization.

   Application ranks will need to select one transaction leader. The leader rank starts and finishes/slips the TID, other ranks can participate in the TID after leader having started it, and the leader should ensure all other ranks have finished I/O operations within this TID before it finishes/slips this TID. As application ranks are process topology aware, they can do fast group collective communication (with possible high-efficient special hardware supporting features) for the transaction status synchronization.

   This is the method similar as DAOS epoch's requirement.

2) IOD internally does the transaction status synchronization.

   Application ranks only need to start and finish/slip this TID separately and independently. For this method, user needs to pass in the number of participators (num_ranks) for this transaction. IOD needs this number to track whether or not all participators have finished this transaction. The "num_ranks" is number of CN-side ranks as function shipper 1:1 forwards/translates I/O calls from CN to ION. IOD will need to do lots of internal P2P message passing for transaction status synchronization. As IOD does not know CN ranks' process group information and function shipping server does not create appropriate process group based on CN ranks' process topology, so IOD cannot use group collective communication. When the number of participators is large, the status synchronizations will introduce considerable overhead and latency at IOD layer.

   A possible optimization exists if IOD can know that this TID is for all CN ranks – we can call it as global transaction. For global transaction, IOD can use the global communicator across all IODs to do similar collective communication by building collective spanning tree to reduce the lots P2P message passing. For this optimization, IOD needs to know two extra parameters: 1) total number of CN ranks and 2) the number of CN ranks which are connected to this IOD. User can pass in these two parameters when calling *iod_initialize*. If application can create dynamic processes, then user should re-call *iod_initialize* when dynamic processes are added. This possibly is a too high requirement to upper layer, so basically IOD can only use lots of P2P message passing for transaction status synchronization.

However, even the higher cost of the multiple P2P messages can be mostly avoided when applications are not extremely asynchronous. Each IOD will count the number of open references for each TID and only communicate with the transaction leader when it sees the TID for the first time and then again when the reference count goes to zero. In the best case, this will be two messages between each IOD and the transaction leader. In the worst case, when no two processes sharing an IOD ever have the transaction open simultaneously, then each IOD will communicate with the transaction leader once to start the transaction and then again for every process participating in that transaction on that IOD. It should be noted that this extreme worse case is expected to be extremely unlikely as this would mean that the asynchrony of the application would be so large that processes would be 1000's of transactions removed from each other. In such an extreme case, the application is encouraged to do its own monitoring of transaction completion as described in method 1.

Applications should be aware of the difference between these methods. Applications which can be synchronous may want to use the first method to minimize cross-talk across IONs. Applications which want fully asynchronous transactions should use the second method.

Besides the possible performance difference, the semantics/usage of method 1) and method 2) have some differences which need to be understood by caller:

- By method 2), application cannot use IOD_TID_UNKNOWN to start a transaction.
- By method 1), application's different process groups can independently participate in same or different TID at the same time; by method 2) different process groups cannot participate in the same TID at the same time.

Start and finish, slip

All processes that want to participate in the TID need to call *iod_trans_start*() with same parameters of TID, number of writers etc. The "num_ranks" is needed for IOD to track that TID's status, zero value means application ensures the status synchronization. Every participator needs to call *iod_trans_finish*() to mark the finish of transaction. The *iod_trans_finish*() can be an asynchronous operation which immediately returns after being submitted to IOD, but the completion of the async-event will be stalled until:

1) All participators of the TID have called *iod_trans_finish*(), and
2) If for writing, all former TIDs become readable/durable or aborted as IOD transactions are ordered.

The slip is like a combination of "finish(old TID) and start(new TID)". The internal ref-count will also be slipped from old TID to new TID.

Abort

The *iod_trans_finish*() can carry an “abort” parameter to abort a transaction, any writer of that transaction can abort it. After successful abort, all writings/updating within this transaction are discarded, IOD will roll back to the status before this TID started.

For writing transaction, user can select two different kinds of abort semantics:

1) Only abort this TID. This means only abort this transaction and will not affect its later transactions. User should select this option if it knows that all higher TIDs have no dependency on this TID.
2) Abort this TID and all higher TIDs that are have been started. User should select this option if the higher TIDs have dependency on this TID. [This mirrors the abort semantics on DAOS.]

User can only abort a transaction before it becomes readable.

Unborn

If a TID is never started, it is unborn. This prevents higher TIDs from becoming readable. This is necessary to allow fully asynchronous applications. Remember that we have two different transaction methods: one, the application can be itself a bit more synchronous and use its own transaction leader, or two, the application can tell IOD how many participants there will be and IOD will provide the transaction leader. In the first method, the application can allow IOD to provide the TID numbers by passing IOD_TID_UNKNOWN. In the second method, the application must itself provide the TID numbering to ensure that different write groups don’t accidentally participate in each other’s transactions. Because of this, we must wait for all lower TIDs to become aborted (independently) or readable before higher TIDs become readable. This is to prevent a situation in which a TID becomes readable and then a slow process tries to start a lower TID.

Consistent semantic

IOD transaction semantic is at container level, after one TID becoming readable on ION user can call *iod_trans_persist* to protect the whole transaction’s state to DAOS. At this point, the TID can be evicted safely from ION since the data is now available on DAOS. After the completion of persisting, the transaction becomes durable on DAOS. That transaction is still kept on ION, IOD will not automatically evict it.

Stale transactions are a bit confusing. On ION, there might be multiple readable TID. If a container is not completely overwritten between TIDs, then the higher TID may rely on data from the lower TID (e.g. the user saved an object at TID=1 and then partially overwrote it at TID=2. When that object is read, some data will come from TID=2 and some will come from TID=1).

Stale transactions are created by a combination of purging from ION and flattening on DAOS. For example, TID 6, 7, 8, 9 are readable on ION, the lowest_durable TID is 6 and latest_rdable TID is 9. At this time point the user might evict 6. Later user persists 9 on DAOS which then renders 6 no longer readable on DAOS. Then 7 and 8 may become unreadable even though they still reside on the ION since any data they rely on from 6 is no longer available anywhere in the system. At this point, they are no longer considered readable on IOD but they may remain useful in case any reads of 9 require data from them. The user must explicitly evict them when desired. [The API does, of course, allow a list of TID's to be evicted in one function call.]

However, the above case in which the persist of 9 causes 7 and 8 to transition from readable to stale is difficult if there are open IOD read handles on 7 or 8. In this case, IOD will ensure the continued readability of those open IOD read handles by ensuring that it has itself an open DAOS read handle on TID=6 on DAOS. That will prevent TID=6 on DAOS from being destroyed when TID=9 overwrites it when IOD does the persist. Note that this is not an ideal performance situation for DAOS as it will need to keep temporary intent logs longer than it would prefer, so we are continuing to explore other ways to preserve the readability of open IOD handles while also allowing DAOS to efficiently flatten multiple transactions.

IOD needs to keep those readable TID's consistent readability. To keep the consistent readability, IOD must ensure the lowest_durable TID on DAOS cannot exceed the lowest readable TID on BB. IOD will follow these rules:

1) At beginning of iod_trans_persist, IOD calls daos_epoch_scope_clone() to get a reference on last DAOS HCE (create a HCE snapshot for it on DAOS); at last step of iod_trans_persist, IOD calls daos_epoch_commit(…, sync, …) to force VOSD to commit this TID without any merge.
2) At 1)'s completion, that TID becomes durable on DAOS. IOD calls daos_epoch_scope_clone() to get a reference of that TID (create a HCE snapshot for it on DAOS).
3) When later user evicts that TID from BB, IOD will call daos_epoch_slip() to release the reference taken at 2).

6.8 Versioning

There is some change from the original SOW in how we will implement versioning. There is no longer any notion of persistent object versioning. This is replaced with temporary views which approximate versioning while multiple transactions are readable on IONs and persistent container level snapshots for data on DAOS.

Using temporary views to approximate object versioning

An approximation of versioning is possible at the IOD layer using transaction id's. Since the user is solely responsible for managing the contents of the burst buffer, they can preserve multiple transactions on the burst buffer. An example of this would be that they can then open handles on these multiple transactions for time-series analysis of the last three state dumps. However, DAOS does automatic flattening of transactions so using transactions as an approximation of versioning on DAOS is more difficult but could potentially be attempted using open epoch handles to temporarily prevent flattening.

Container snapshots

Permanent versions can be created into the namespace with DAOS container snapshots which are a space-efficient, copy-on-write mechanism for entire container snapshots. This is like object versioning as proposed in the original SOW except:

- It is for containers and not single objects. A user wanting snapshots on a single object could, of course, create a container with only a single object.

- The namespace is a bit different. Instead of version ID's on a single entry, this creates multiple entries within the namespace.
- This will only be done on the DAOS layer. A user wanting snapshots must first migrate the container, and then snapshot it. Users wanting versioning at the ION layer can use the transaction scheme as described above and can also migrate and snapshot each of the views thereby achieving both the temporary versioning approximation and the permanent one. This will work for streaming data.

6.9 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.

IOD API has parameters for checksums to be passed to and from its upper layer which it stores alongside the data as additional metadata. These checksums can be passed all the way to the application to provide full end-to-end data integrity protection against silent data corruption. Partial reads will of course require IOD to read the full chunk, check the integrity, and then create a new checksum for the partial read. HDF will link with IOD and share the checksumming function.

Note that this end-to-end integrity will not reconstruct corrupted data; merely it is designed to prevent silent data corruption. In other words, corrupted data will always be detected but not reconstructed.

With the development of resource virtualization, the application becomes farther from the storage hardware. And there are more layers between the application and the hardware, and if an error happens inside those middle layers, the application may get the wrong data back and it may even not notice that the data is wrong. This is something we do want to avoid. The basic idea to verify the data is that all data will be associated with a checksum, so when the checksum mismatch the data, we could know that a data corruption has occurred.

The application will need to do the following things. For write, calculate and associate a checksum for the data send to the IO middleware either directly to IOD or though HDF. Note that the application may choose not to participate in checksumming data but request that IOD do it instead. These provides better protection that in the current POSIX stack but not quite as good as true end-to-end as if the application actively participates. For reads, IOD will return the data together with a checksum, verify it.

IOD will move data to the storage system with some optimizations to get better performance.

Some terminology for the following discussion:

Buffer represents a piece of user's data which has or will be given an associated checksum. It's the basic unit handled by our algorithm. A **region** is a part of a buffer. IOD only stores/retrieves data between application and storage, so all the buffers generated inside the IOD are derived from the buffers from application or storage using the following 4 operations:

Copy: copy the data from a target buffer to a new buffer.

Slice: copy a region of a target buffer into a new buffer.

Merge: copy several targets buffers into a new buffer in a specific order. For example, the user is doing a large aligned read that spans multiple buffers on storage.

Slice&Merge: copy regions from several target buffers into a new buffer in a specific order. For example, the user is doing an unaligned read that spans the second half of one buffer and the first half of another.

This document describes how to use the checksums of the target buffers to ensure that the checksum of the new buffer is correct.

It is important to understand the checksum algorithm which we are using. It works as an accumulator such that you can give it multiple buffers individually and it will accumulate a checksum value incrementally for each one. The checksum value returned is identical to the value that would be returned if the multiple buffers were concatenated into one contiguous region and input into the checksum algorithm.

Here are the main functions provided by the checksum algorithm:

```
int mchecksum_init(const char *hash_method, mchecksum_object_t *checksum);

int mchecksum_get(mchecksum_object_t checksum, void *buf, size_t size, int finalize);

int mchecksum_update(mchecksum_object_t checksum, const void *data, size_t size);
```

Init creates a new empty checksum object. Update increments it by giving it a new data region. Get can be used to finalize it and get the checksum back as a bunch of bytes.

For each of the four operations, we define the following algorithms which accept a set of one or more source buffers or regions and return one contiguous buffer and an associated checksum. Each source buffer has its own checksum which we refer to as a source checksum. These operations happen every time data is moved through IOD.

1. Copy the source checksum into the return checksum
2. Verify the source checksum and source buffer
3. Copy the source buffer into the return buffer

Note that this ordering prevents silent data corruption no matter when it occurs. For example, if the source buffer is corrupted between steps #2 and #3, then the return buffer will contain the corrupted data. However, since we copied the return checksum before verifying the source buffer, the corrupted data will not match the checksum and the corruption will be correctly detected. The other operations are more complex but provide this same guarantee.

Here are the steps for a slice operation.

1. Calculate return checksum from the source region of the source buffer
2. Verify the source checksum against the source buffer
3. Copy the source region from the source buffer into the return buffer
4. Return the return buffer and return checksum

Note that the return checksum is always calculated based on the data from the source buffer, and then verify the source buffer. The order is important so that if data corruption happens, the return buffer and return checksum will not match or the source buffer will fail the verification. In this way, there won't be any silent data corruption.

Here are the steps for a merge operation.

1. Create an empty return checksum
2. For every source buffer (ordered correctly), repeat the following steps 3 - 5.
3. Accumulate the source buffer into the return checksum
4. Verify the source buffer with its checksum
5. Copy the source buffer into its appropriate location in the return buffer.
6. Return the return buffer and return checksum

Here is pseudo-code for this operation, suppose that we want to construct a new buffer (retbuf, retchk) from a list of source buffers [(buf1, chk1), (buf2, chk2), (buf3, chk3)]. Here is the code how the retchk and retbuf is composed:

```
mchecksum_init("crc64", &tmpchk);
for (buf, chk) in [(buf1, chk1), (buf2, chk2), (buf3, chk3)] {
    mchecksum_update(tmpchk, &buf, sizeof(buf)); // COMMENT ONE
    if (verify_checksum(buf, chk) == FAILED) return –EIO;  // COMMENT TWO
    memcpy(retbuf + offset_i, buf, sizeof(buf));
}
mchecksum_get(tmpchk, &retchk, sizeof(retchk), 1);
return (retbuf, retchk);
```

Note the ordering in the pseudo-code of the lines marked 'COMMENT ONE' and 'COMMENT TWO.' This ordering is important to prevent a race condition in which the data can be corrupted between the copy and verify operations.

Here are the steps in a slice and merge operation.

1. Create an empty return checksum
2. For every source region (ordered correctly), repeat the following steps 3 - 5.
3. Accumulate the source region of the source buffer into the return checksum
4. Verify the entire source buffer with its checksum
5. Copy the region of the source buffer into its appropriate location in the return buffer.
6. Return the return buffer and return checksum

This a combination of the Slice operation and Merge operation. Based on these four basic operations, we can produce a new buffer with the right checksum, based on several source buffers and then verify the checksums associated with them.

For the storage layer, it must store the checksum into the storage device so that the whole solution could provide end-to-end data integrity. So that when the data is read back in the future, the application can still be confident that the data is not corrupted by checking the checksum which was persistently stored.

When IOD stores data into DAOS, it stripes that data across a set of DAOS objects which we refer to as data objects. Each data object has an associated checksum object. And the DAOS objects are divided into fixed-sized chunks, each of which has a checksum stored in the checksum object. For chunk N in the data object, its checksum is stored at offset of (N * sizeof(checksum_t)) at the checksum object.

Every modification to a chunk will cause an update to the whole chunk and the checksum so that we can keep the data and the checksum consistent. The following sections describe how to handle write/read operations for DAOS objects.

To write to DAOS, only two cases are possible:

1. IOD needs to write a full checksum chunk.
2. IOD needs to write part of a checksum chunk.

Then all the write operations can be treated as a combination of the above two writes, so that if the above cases are handled correctly, we can handle all the write operations correctly as well.

For full write, here is the pseudo code:

```
int iod_daos_write(iod objectid, source buffer, offset, length, source checksum)
{
        data_oh = open_daos_data_object(iod objectid);
        checksum_oh = open_daos_checksum_object(iod objectid);
        checksum_offset = (offset / chunk_size) * sizeof(checksum_t);
        daos_object_write(data_oh, source buffer, offset, length);
        daos_object_write(checksum_oh, source checksum, checksum_offset, sizeof(checksum_t);
        close all daos objects;
}
```

For partial write, the code is more complicated:

```
int iod_daos_write(iod objectid, source buffer, offset, length, source checksum)
{
        mchecksum_init("crc64", &tmp_chksum);
        data_oh = open_daos_data_object(iod objectid);
        checksum_oh = open_daos_checksum_object(iod objectid);
        chunk_offset = round_down(offset); // get the start of the whole chunk;
        checksum_offset = (offset / chunk_size) * sizeof(checksum_t);
        original_chunk = malloc(chunk_size);
        daos_object_read(data_oh, original_chunk, chunk_offset, chunk_size);
        mchecksum_update(tmp_chksum, original_chunk, offset – chunk_offset);
        mchecksum_update(tmp_chksum, source buffer, length);
        sourceend = offset + length – chunk_offset;
        mchecksum_update(tmp_chksum, original_chunk[sourceend], chunk_size – sourceend);
        mchecksum_get(tmp_chksum, checksum, sizeof(checksum_t), 1); // Comment 1
        daos_object_read(checksum_oh, original_checksum, sizeof(checksum_t));
        mchecksum_verify(original_chunk, original_checksum); // Comment 2
        mchecksum_verify(source buffer, source checksum); // Comment 3
        daos_object_write(data_oh, source buffer, offset, length);
        daos_object_write(checksum_oh, checksum, checksum_offset, sizeof(checksum_t);
        free(original_chunk);
        close all daos objects;
}
```

As noted before, we must verify the buffer after using it. So we verify the chunk read from the data object and the source buffer after the new checksum is constructed. Please refer to comments 1, 2, and 3 in the pseudo code above.

Read operation is similar to the write operation in which only full read and partial read need to be considered. However the handling of partial read is much easier than the partial write.

Here is the pseudo code for full chunk read:

```
int iod_daos_read(iod objectid, return buffer, offset, length, return checksum)
{
        data_oh = open_daos_data_object(iod objectid);
        checksum_oh = open_daos_checksum_object(iod objectid);
        checksum_offset = (offset / chunk_size) * sizeof(checksum_t);
        daos_object_read(data_oh, return buffer, offset, length);
        daos_object_read(checksum_oh, return checksum, checksum_offset, sizeof(checksum_t);
        close all daos objects;
}
```

For partial read, the code is as following:

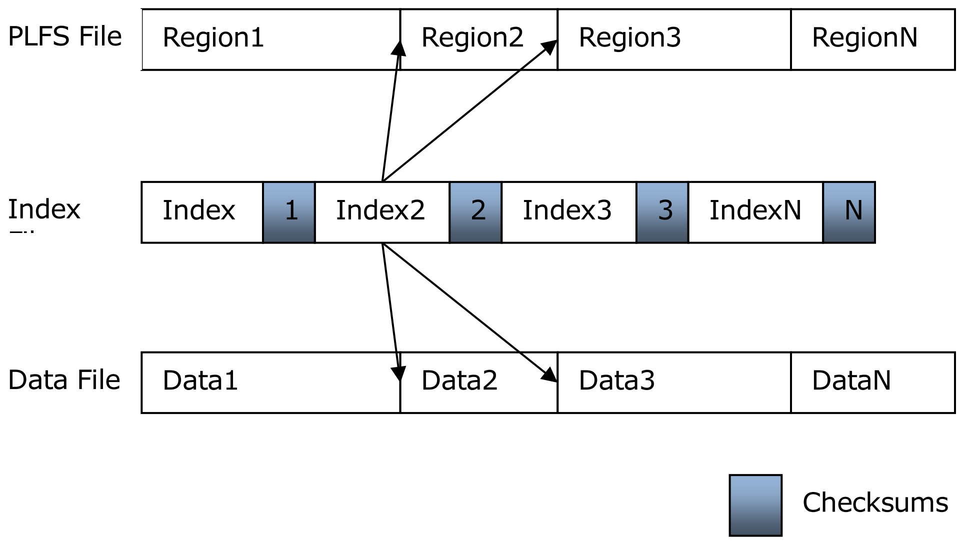
```
int iod_daos_read(iod objectid, return buffer, offset, length, return checksum)
{
        data_oh = open_daos_data_object(iod objectid);
        checksum_oh = open_daos_checksum_object(iod objectid);
        chunk_offset = round_down(offset); // get the start of the whole chunk;
        checksum_offset = (offset / chunk_size) * sizeof(checksum_t);
        original_chunk = malloc(chunk_size);
        daos_object_read(data_oh, original_chunk, chunk_offset, chunk_size);
        daos_object_read(checksum_oh, original checksum, checksum_offset, sizeof(checksum_t);
        // Apply Slice operation defined in the previous chapter.
        return_buffer_offset = offset – chunk_offset;
        mchecksum_init(“crc64”, &tmp_chksum);
        mchecksum_update(tmp_chksum, original_chunk[return_buffer_offset], length);
        mchecksum_get(tmp_chksum, &return checksum, sizeof(checksum_t));
        mchecksum_verify(original_chunk, original checksum);
        memcpy(return buffer, &original_chunk[return_buffer_offset], length);
        free(original_chunk);
        close all daos objects;
}
```

In this way, read and write to DAOS can be handled correctly with checksums to prevent silent data corruption. Checksums can be stored, updated or retrieved from the DAOS storage system successfully.

Storing data into the burst buffers will require a different data integrity implementation as DAOS is not (currently) exporting the burst buffers. Instead IOD uses PLFS, in which the data to be written from different nodes will be appended into different physical files to get the maximum performance. The possible conflicts between different write operations will be resolved according to the timestamp when the application wants to read the data back. So we will now describe how to handle the checksum for PLFS write and read operations.

For every write, the data is appended to the data file, and the mapping information of the logical data and the physical data in the data file is appended into an index file. The data in the data file is supplied by the application and the entries in the index file is generated by PLFS to track the write so that it can read the data back later.

The following figure shows how the index entry works.



So every index entry will map a segment in the logical PLFS file to a segment in a data file and the timestamp information so that conflicts can be resolved.

For checksum, we will append the checksum of the written data to the index entry associated with this write operation as shown in the figure. In this way, with the checksum feature enabled, we can still get almost 100% performance for write operations, since the only thing we need to do is verifying the checksum and write the checksums into the index file. Since the checksum verification is fast and the checksums are much smaller compared to the data, the performance should be good.

There is one possible optimization for write, in which we can disallow the write request from covering a very large segment. Because that when the data is read from the data file, it will be verified as a single object, so if we write a very large segment, we will always need to read the whole segment back to verify its correctness whenever even a very small part of it is accessed. 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. Therefore, IOD currently imposes a 4 MB maximum write into PLFS; this value is configurable.

For read operations, all the index entries in those index files are loaded into memory to construct the global index tree. So that for every piece of data in the PLFS logical file, PLFS can find where the data is in the data files.

Considering that the written data might overlap with each other, maybe only parts of the original data written is still valid in this PLFS logical file, the other parts might have already been over written. The previous implementation of PLFS will simply discard those invalid parts, however we can’t do this for checksums, so we had to reimplement PLFS to handle this correctly.

The checksum is calculated based on the whole buffer in the write operation, so we must read the whole buffer as the write operation so that we can verify it against the checksum in the index entry. This requirement changes the way we construct the index tree and perform the read operations.

So we must treat all the index entries as a whole object and we can’t split them or merge them. So the PLFS index code has been revised and split it to three layers:

- Physical Entries: These are entries loaded directly from the index file. Because of checksums; we need to treat it as a single object without splitting or merging.
- Remap Entries: These are entries in the global index tree. It will map a segment in the PLFS logical file to a whole/part of the physical entry.
- Index Tree: A map from logical offset to the Remap Entries, so that we can find mapping information of a given segment in the PLFS logical file. The tree contains non-overlapped logical segments of the PLFS logical file.

Please refer to the following figure for more information:

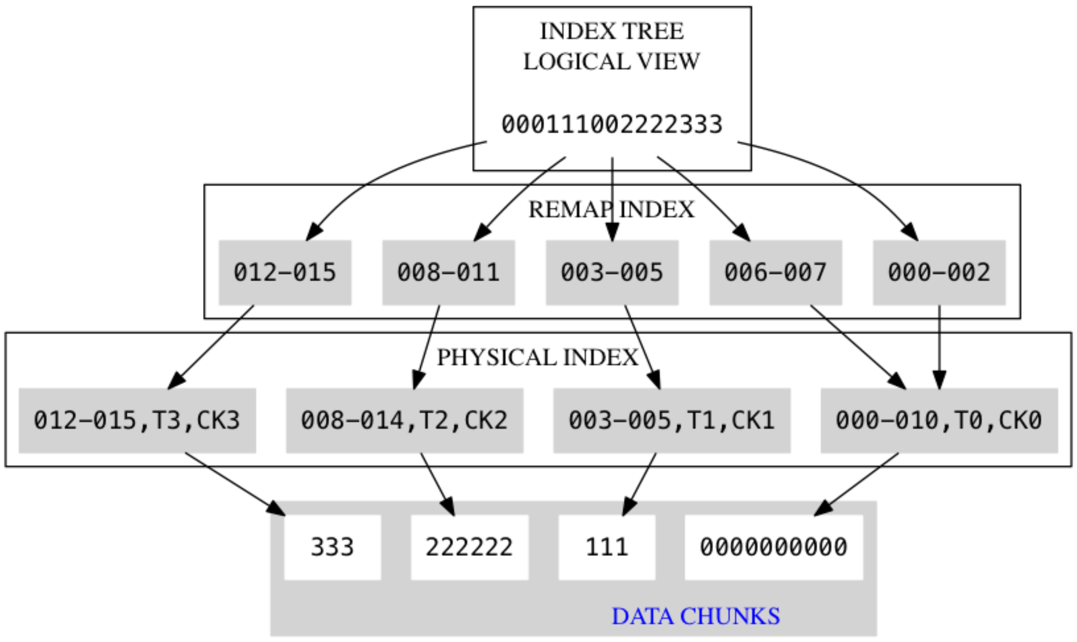


Figure 11 Index Tree with Checksums

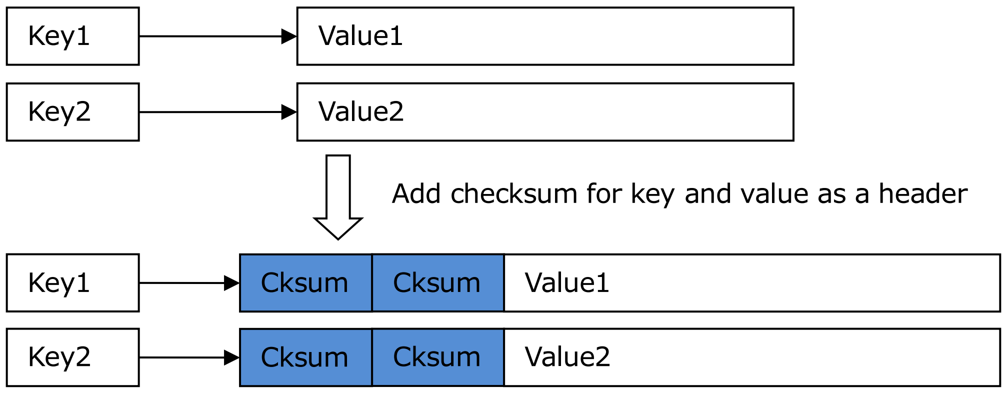
For read tasks, thread pool will be used to improve the performance. Accordingly, the read will be divided into 5 steps:

1. Find all the physical entries for the given logical segment, and record the remap entries so that we can know the mapping between physical data and the PLFS logical file. This will be done in the main thread since it's pure memory operations.
2. Read all the physical entries into memory in parallel.
3. Accumulate the checksum for the return buffer in order. Note that the order is important to the correctness of the checksum.
4. Verify all the physical entries and its checksum. This can be done in parallel as an optimization.
5. Copy the data to its destination if needed. This only needs to be done if only part of the physical entry is included in the return buffer. See next section for more details.

For step 2, If the whole physical entry is still valid and covered by the logical segment, then we will use the buffer supplied by the user to store the data read from the data file, so that the memory copy in step 5 can be avoided. However if only parts of the physical entry is valid or covered by the logical segment, then we must allocate a new buffer to hold the whole physical entry and we must move the data to the right place in step 5. Using our earlier terminology, copy and merge do not require the memory copy but slice and slice/merge do.

In this way, we can handle the checksum in plfs_read() and plfs_write(), and it only add a very small overhead to PLFS, if there is no data overlap and partial read, then the overhead is only caused by checksum verification and calculation, which is unavoidable.

The above descriptions describe how data integrity is implemented for blob and array objects. However, KV objects must be treated differently. Since K-V checksum would be associated with the whole K-V value, so it won't face the problem that PLFS or DAOS have since it will always only use the copy operation and never the merge, slice, or slice&merge. The current design will be to have the key checksum and the value checksum be a header at the front of the value. The following figure shows the change we made to it:



Since the API for K-V store all works on the whole key-value pair, so that we can always get/update the header together with the key-value pair, and there is no need to do the slice or merge operations for it.

6.10 Asynchronous operation and event

IOD strives for asynchrony to allow user can build fully non-blocking applications. One IOD API's success return just means the request has been submitted to IOD, a related completion event can be polled by user when it finally finishes executing.

For event queue (EQ) and event:

- A queue that contains events inside, user can create event queue at any time and allocate/initialize an event to bond it to one EQ.
- Events are used by all asynchronous IOD APIs. Most IOD APIs are asynchronous (except API to create EQ, initialize event…).
- User can register a callback to the event, so later after that event finishes, the callback will be triggered.
- Event queue (EQ) and events are used for tracking completion event of IOD functions.
    - IOD function can return immediately only means that request has been submitted to IOD but doesn't mean it has completed, the only way to know completion of operation is polling completion event.

    - If caller passes in NULL event pointer then means synchronous call, the caller will be blocked until finish.

IOD's async-engine is the center for executing all asynchronous operations. When an asynchronous request is received, IOD will insert it to an appropriate task list and bond the task with the async-event. The IOD I/O scheduler will pick up the task and use thread to execute it.

6.11 Impact on HDF5 users

- IOD makes some extensions/restrictions to HDF5 dataset:
    - The dataset is one dimension extendable, can only be extended along the first dimension, i.e. all other dimensions must have fixed length.

      This is same as HDF5's original external dataset except that it needs not to be stored by external dataset files. IOD will store it in one array object within the container.
    - IOD adds supporting of changing the layout mapping between logical dimensions to physical dimensions sequence. This is the basic idea for semantic resharding.

      For extendable array, the first logical dimension must also be the first physical dimension – to make the address space be calculable.
- Transaction is the basic unit of data migration/purging/replica. User should control the reasonable transaction granularity.

7 Configuration Parameters

IOD has many important configuration parameters that we have seen to be very important to extract maximum performance from the system. Unfortunately, they do currently need to be tuned manually although we have attempted to set reasonable default parameters.

Note that the parameters which control chunk and block sizes such as the checksum chunk sizes and the daos striping sizes are all adjusted internally to be multiples of each other as well as the array cell size for IOD array objects.

WHICH	WHAT	DEFAULT
iodrc: **iod_daos_stripe_mbs**	When large objects are stored on DAOS, they are stored in a RAID0 striping. This controls the stripe size.	128 M
iodrc: **iod_threadpool_size**	Thread pool size	16
iodrc:	How many shards will be created on a single DAOS storage target.	1

iod_shards_per_target	For small DAOS installations, it is best to adjust this so that total IONS == total shards. Otherwise, just one per target.	
iodrc: **iod_checksum_chunk_bytes**	The checksum chunk size on DAOS.	1M
iodrc: **central_io_buffering**	Turn on DAOS IO buffering, 1 for turn on and 0 for turn off (default). Caution: Don't turn it on now because there are some bugs related to DAOS bugs.	0
iodrc: **iod_max_persist_memory**	The maximum size of the IOD memory buffers used when migrating data for replicas, fetches, and persists.	128MB
plfsrc: **max_index_length**	It should be multiple of iod_checksum_chunk_bytes and should not be too large. This is the size of the checksum chunk in PLFS. Analogous to iod_checksum_chunk_bytes	8MB
plfsrc: **threadpool_size**	Number of PLFS threads used to build global index on a read.	8
plfsrc: **mlog_setmasks**	This controls how much debugging. Should be ERR unless debugging.	ERR
plfsrc: **backend IO interface**	PLFS backends can be prefixed with 'posix:' or 'glib:' to control whether PLFS internally uses the non-buffering <fcntl.h> routines or the user-space buffering <stdio.h> routines. 'glib:' is much better for small IO's less than 1MB; 'posix' slightly better for larger.	'posix:'

8 Performance Evaluation

The following graphs demonstrate the current performance of IOD running on the Cray buffy system at LANL as well as the Intel lola test system as of June 2014. Some graphs show performance as expected whereas others reveal areas in need of future work. We have considered carefully the problem areas and are convinced that all of them can be well-addressed with software optimizations and are not reflective of design flaws.

Note that these performance evaluations already have addressed several areas which were in need of improvements which have been implemented as of this time:

- Reduce overly-eager PLFS index merging
    - PLFS will extend index entries indefinitely for contiguous data
    - PLFS has one checksum per index entry
    - Small reads within a giant checksum chunk are very slow
- Better aggregation and larger writes to DAOS during persists
- Using <stdio.h> FILE * IO instead of <fcntl.h> IO
    - Provides client side user-space buffering and caching
- Many performance tuning parameters in iodrc and plfsrc
    - Threadpool sizes
    - Stripe sizes
    - Checksum chunk sizes
    - Memory consumption during persist
    - DAOS shards used per DAOS storage target
- Incremental KV persist
- Optimized sorting in range queries of data across multiple TIDs

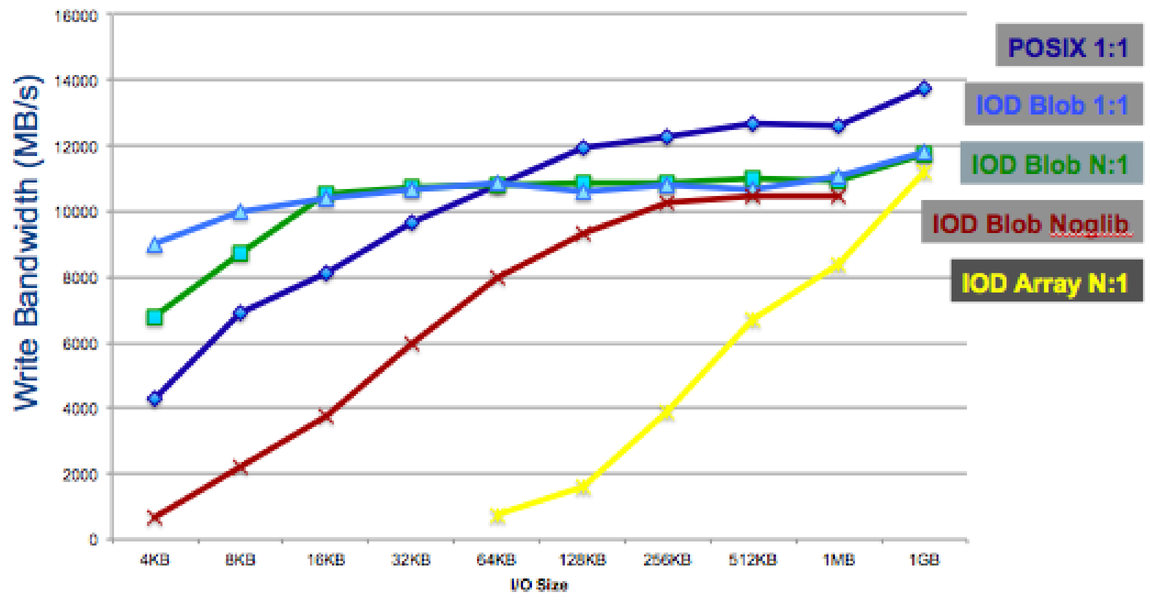


Figure 12. This figure shows that IOD blob performance for both shared file and file-per-process perform mostly as expected. The slight dip in performance for shared-file blobs for small-IO sizes is an open question as is the slower performance for array objects. This data was collected with IOR running on buffy; the POSIX measurement was taken using a local file system on each node. The "IOD Blob Noglib" measurement shows the value of using the C functions for IO in the stdio.h library instead of the C functions in the fcntly.h library since they add important user-space buffering.

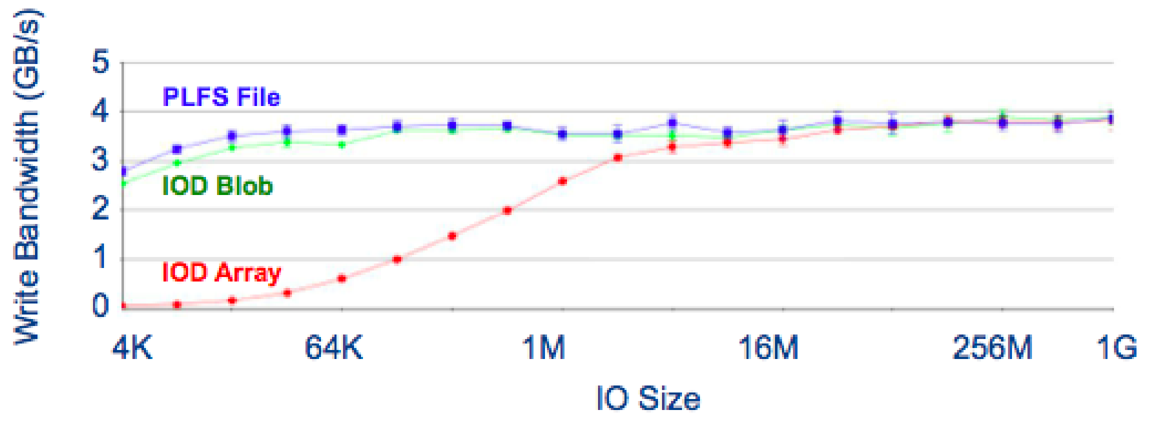


Figure 13. This figure compares blob and array write performance to PLFS. Since IOD is implemented using PLFS to store writes, it is a satisfying result that blob IO only incurs a slight degradation over PLFS. IOD arrays are under current investigation. This data was collected using the LANL fs_test benchmark running on lola. Note that these tests set the array cell size equal to the IO size. However, in other IOR measurements, not shown, we set the array cells to be a constant 8 bytes regardless of the IO size and observed that this did not affect performance.

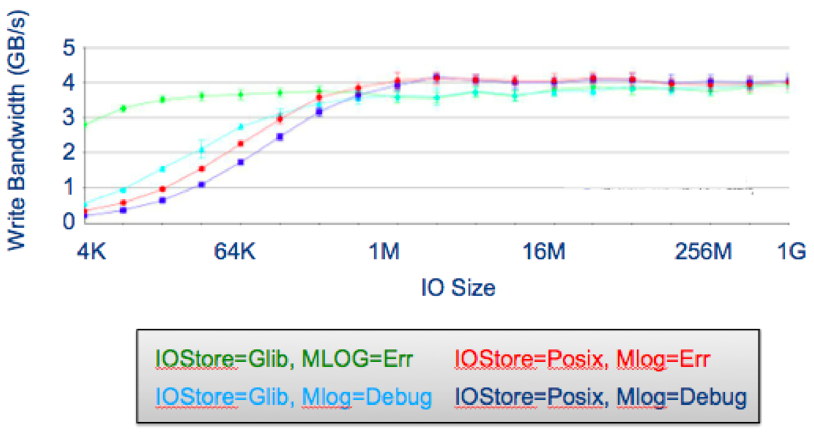


Figure 14. This figure shows the importance of handling small IO appropriately. The value of user-space buffering is shown with the lines marked "Glib" which indicates that the IO was performed into the BB's using the stdio.h routines; lines marked with POSIX used the routines in fcntl.h. Lines marked with Err and Debug respectively show the extreme damage to small IO performance which can be caused even with small perturbances such as a debug output message sent to the /tmp partition.

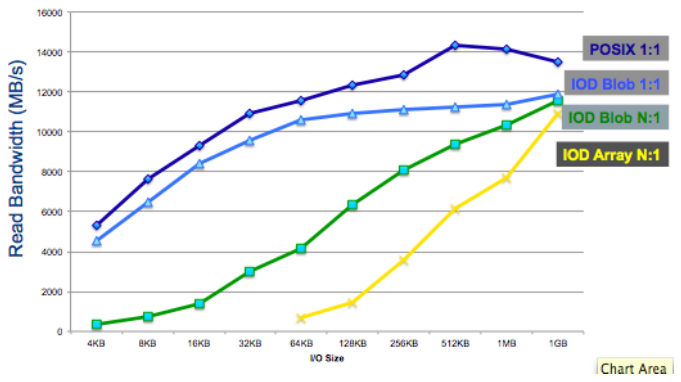


Figure 15. Just as in writes, the blob performance for file-per-process is close to the upper bound of the POSIX bandwidth. The single-shared-file performance is lower due to the slow open time during which PLFS consolidates the large amount of metadata from the multiple PLFS index files. Note that the LANL PLFS team is working on several optimizations for this challenge. As in the writes, the array performance for small IO is unexpectedly low.

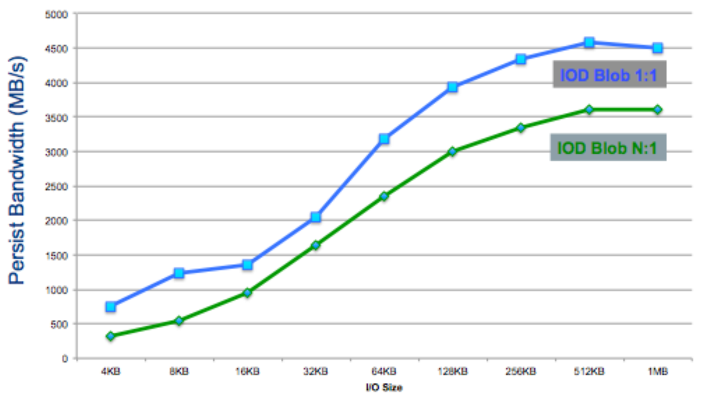


Figure 16. The performance to persist an IOD blob from BB to DAOS is shown here. The difference between the file-per-process and single-shared file may be due to the slower time to read the single-shared file from the BB's due to the aforementioned PLFS metadata. It may also be due to reduced parallelism while writing to the DAOS shards.

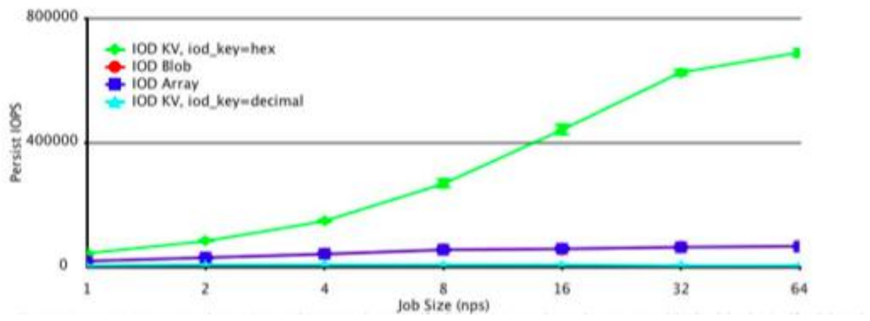


Figure 17. This picture shows that IOD KV's can be persisted to DAOS at a higher IOPs rate than blobs and arrays when the key is distributed across the range servers. Note that the performance for decimal key's is very low. This is because the decimal keys all were sharded to a single range server within the MDHIM table and therefore parallelism was not achieved. With a hexadecimal key more uniformly distributed across the key space, the IOPs scale with the size of the job.

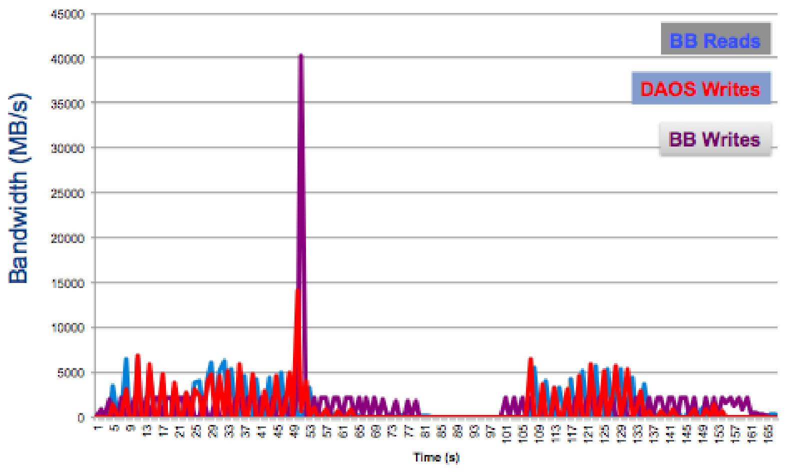


Figure 18. This graph shows storage activity during a workload in which an application was writing new transactions into the burst buffers while IOD was persisting an older transaction to DAOS. IOD mostly is able to overlap IO although the strange gap in the middle needs investigation, as does the strange 40 GB/s spike.

9 Lessons Learned

Many lessons learned and recommendations for future work can be found in the overall FF Final Design document, the *EFF0* document, available by request from ira.lewis@intel.com or john.bent@emc.com. Additionally, there are many items for IOD future work that became evident during the course of the project.

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[1].

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.

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

[1] 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

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).

IOD does not currently checksum its own metadata and not all IOD functions, such as fetch, have list variants. Pipelining the checksum computations with IO is something that is not currently done and probably should be.

As we move forward with this project, it may become necessary to revisit some early design decisions. For example, at the beginning of the project, we decided that we would not have a resilient burst buffer layer. This seems to be mostly a reasonable decision especially when the burst buffer is used for checkpoint restart. Since the number of expected burst buffer nodes is approximately two orders of magnitude lower that the number of expected compute nodes, the likelihood of concurrent failure of both the compute and the burst buffer layer is very low. Therefore, the cost of resilience is greater than the benefit. Therefore, the original design maintained all IOD metadata in memory thereby preventing recovery of ION data following an IOD crash. However, the actual implementation relaxed this design and kept much of its metadata persistently. This might be the right decision both because it allows resilient burst buffers that can recover following failure (although not necessarily reconstruct lost data) as well as because it reduces the need for complex memory management with the IOD software.

Additionally, a systematic review should be done across all routines in IOD to determine which might reduce asynchrony. For example, currently all processes using IOD must barrier before calling iod_container_close. Another example is that only one process can call iod_obj_fetch and then broadcast the replica tag before other processes can ust that prefetched object. Note that this limit on synchrony will be relaxed with the addition of *tagless replicas*.

Tagless replicas, described in great depth in the EFF0 document, refer to the current IOD semantic in that reads of replicas must use a replica tag which is returned to the caller when the caller requests a fetch. This is suboptimal because the user must track which replica tag to use for each read. Preferably, IOD should allow all reads to merely pass a transaction ID and IOD will route each request to the most local replica. Additionally, the tagged replicas fall outside the normal transactional semantic. For example, if the user fetches TID=4 into tagged replica 4.1 and then does partial overwrites into TID=5, then after TID=5 is committed, reads of holes on TID=5 are resolved by IOD by reading the data from TID=4 on DAOS instead of from the more local tagged replica 4.1.

Additionally, some routines, such as iod_obj_purge and iod_obj_persist, take an object handle as a parameter whereas they might be better suited taking an object ID as that doesn't require that the object be opened. Indeed, the original design decision to require open object handles should be revisited as the open transaction is effectively a handle already. Early in the project, it was thought that IOD might support named objects in which case a broadcastable object handle would have been useful. However, HDF did not need named objects so we did not end up supporting them. Therefore, an IOD object ID is easily broadcastable and, as previously said, their protection is provided by the transaction mechanism.

10 Conclusion

The basic IOD data type definitions and API are available on the public wiki at https://wiki.hpdd.intel.com/display/PUB/Fast+Forward+Storage+and+IO+Program+Documents. A version of IOR with DAOS, HDF, and IOD modules is available here: https://github.com/johnbent/ior. LANL fs_test with an IOD module is available here: https://github.com/johnbent/fs_test/tree/iod.

The design and implementation of the IOD software to enable exascale storage via the inclusion of a hardware burst buffer and a new software API has been challenging and instructive. Valuable feedback was gained through cooperative development with the HDF team building an exascale version of HDF5 as well as with the Intel team building DAOS and the Intel team building an acyclic graph big data analysis problem to demonstrate the value of the full exascale stack. Although we cannot be sure whether IOD will be the software running on exascale HPC supercomputers, we are confident that many of the design decisions and features are necessary, well-defined, and well-tested, and will be included in whatever software ultimately is used.

11 FAQ

1. Reader processes that share the same IOD instance as writer processes (through HDF VOL for example) can use the same transaction #'s to see particular container states (views)
    - Correct. Although note that, in our HDF-IOD-DAOS stack, there aren't exactly IOD instances per se but rather IOD library is linked into the VOL server instances.
2. Reader processes that don't share the same IOD instance (either they run on different systems but share DAOS or run at different times) can't use the

original transaction #'s to see a particular container state. They will be 'given' the latest HCE when the open the container, and told it is 0.

  - Correct. Although I think that DAOS will change this to provide absolute HCE not relative which I think will be more useful. Without this, then I will push for IOD to maintain a mapping between the relative that DAOS provides and the absolute which I think will be more meaningful to IOD's upper layer.
  - Note a very important point. DAOS provides a write lock on the container so IOD does not. Whenever a process group tries to get a write handle on an IOD container, IOD tries to get a write handle on the corresponding DAOS container. In this way, there can be only one process group writing to a container. This is extremely important because if we did allow multiple process groups to write to a container, then the transaction IDs (epoch IDs) would become a mess.
  - But a write group to a IOD container can persist transactions to DAOS where those transactions will then be available to an independent read group.

3. There will be mechanisms provided by DAOS to "get next" or "go to end"
   - Yes, and IOD as well. For DAOS, it is referred to as HCE and you can "slip" to get to it.
4. File contents can be used to coordinate between data producers and consumers that do not share the same IOD instance, for example, a "last checkpoint" attribute & writing checkpoints to separate groups rather than over-writing.
   - Correct. This would be a scenario like: write group starts TID=3; write group writes {x,y,z}; write group ends TID=3; write group persists TID=3; read group opens DAOS HCE; read group reads {x,y,z}
   - Notice that the data has to go back and forth to DAOS. Conversely f the write and read group *are* connected to the same IOD daemon processes, then they can produce-consume on the IONs. To demonstrate this, we could do something like launch a large MPI job and split MPI_COMM_WORLD into MPI_COMM_SIMULATION, MPI_COMM_VOL, MPI_COMM_IOD, and MPI_COMM_ANALYSIS. But I don't know if they will actually ever be demonstrated.
5. If a reader has a handle for a given view, they are guaranteed that they can continue to see the contents of the container for that view until they release the handle. Attempts to evict data in transactions that are needed to serve this view will fail.
   - Yes. By the way, remember there is also the weird scenario where persist can destroy readability of earlier TIDs. So persist might also fail if it would destroy the readability of an open read handle. For example:
   - TID=1, TID=2, TID=3 are readable in IONs. TIDs 2 and 3 did partial overwrites so they rely on data from earlier TIDs; The user has a read handle on 2; The user persists 1 and evicts 1; 2 is still readable because any missing data from TID=1 can be read from DAOS; The user tries to persist 3. IOD can either: Say no because persisting 3 may cause DAOS to flatten it over 1 thereby destroying any data in 1 which might be needed for 2 –or- Grab a read handle on 1 on DAOS

and then persist 3 because the read handle on DAOS will prevent the flattening

6. Evictions don't wipe out the whole transaction. They say, in effect, "evict the object or partial object I've stenciled from the BB as of a particular view of the container, denoted by this TID"
   - Yes. Evict and pre-stage are opposites except that pre-stage can operate on sub-objects whereas evict operates on all data written to the object within that transaction.
   - Pre-stage also will have the notion of a "bundle" where it can ask IOD to pre-stage sub-objects {x,y,z} and place them together on a single ION and then bundle sub-objects {a,b,c} and place them together on a different ION. This is when analysis knows that one of its tasks needs {x,y,z} and another needs {a,b,c}.
   - To be clear, evict and pre-stage are not exactly opposites since pre-stage operates on sub-objects@TID whereas evict operates on entire objects@TID. This is because the data for an object@TID on IONs is scattered across multiple PLFS style logs and to evict sub-objects would require copying out the sub-objects which aren't evicted. If the users wants to only evict sub-objects and preserve the rest of the objects, then they need first to do a multi-format replica or a semantic resharding and then do the eviction.
7. New opens of a container that ask for the latest HCE get '0' from DAOS if they are not part of an IOD instance that already has the container open. If they are, then they get the HCE TID of the pre-existing open [is this true]
   - I think DAOS will provide an absolute epoch. At one point, they discussed relative epochs always restarting at 0 for each open but I believe they have now decided to maintain an absolute epoch.
8. Transactions can be started in any order.
   - True for IOD transactions.
9. Transactions may be finished (as in application calls HDF5_transaction_finish which calls iod_trans_finish) in any order.
   - True for IOD transactions.
10. Transactions will become readable in increasing (possibly with gaps) order as Transactions are finished
    - True for IOD transactions so long as gaps are caused by aborting a transaction which is specified to be an independent transaction. But if there is some transaction which was never started or is still open, then future transactions cannot become readable.
    - An independent transaction just means that the user does not want its abortion to cascade and abort higher transactions.
11. Regarding whether transactions are persisted in a transaction, it looks to me like iod_trans_persist takes a tid but is not called *in* a transaction. Maybe it's just 2 perspectives on what that TID means.
    - Agreed. However, persist will use DAOS transactions to ensure that the persist, which may be a large number of operations across a large number of shards, is atomic. So when the user says, "persist TID=3," then IOD will use epoch=3 to do the migration so that the view on ION of TID=3 is the same as the view on DAOS for epoch=3.
    - Note that we sometimes say "migrate" instead of "persist," but "persist" is a more accurate term since "migrate" suggests that the

data may be moved out of the original location but this is not what happens when a user "persists" IOD transactions. "persist" is basically a 'cp' and not a 'mv.'

12. You can't make a multi-format replica on an array object???
    - multi-format replica and semantic resharding are the same thing except that semantic resharding is for array objects and multi-format replica is for blobs (and maybe kv's). That's why you can't do a multi-format replica on an array object; you can only do a semantic resharding.
13. What is difference between "migrate" and "persist?"
    - We tend to use them interchangeably but really we should only say "persist." When we use them, we just mean that we take the view of the container at TID=t and make it persistent on DAOS so that reads of TID=t from ION and reads from DAOS will return the same data. "persist" is the better term however since we do not remove the data from the original location as you might expect from the word "migrate." It is analogous to POSIX 'cp' not POSIX 'mv'
14. Can you explain your transaction semantics again please? Also, it seems like they operate different for reads and writes.
    - There are two key protections that our transactions provide:
        i. Read protection.
        ii. Write protection.

12 References

[1] John Bent, "IOD solution architecture", Fast forward internal document.

[2] John Bent, etc., "PLFS: A Checkpoint Filesystem for Parallel Applications", in Proceedings of SC09, Nov. 2009.

[3] Zhenhua Zhang, "IOD KV store high level design", Fast forward internal document.

[4] Eric Barton, "DAOS solution architecture", Fast forward internal document.

[5] Zhen Liang, "DAOS API and DAOS POSIX design", Fast forward internal document.

[6] Jerome Soumagne, etc., "Function Shipping Design & Framework Demonstration", Fast forward internal document.

[7] Quincey Koziol, "HDF5 solution architecture", Fast forward internal document.

[8] Quincey Koziol, "Indexing Chunked HDF5 Datasets with One Unlimited Dimension", http://www.hdfgroup.uiuc.edu/RFC/RFCs/HDF5/ReviseChunks/skip_lists/SkipListChunkIndex.html.

[9] HDF5 document, "Chunking in HDF5", http://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/.

[10] Paul Nowoczynski, “VOSD solution architecture”, Fast forward internal document.

[11] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn,and Xian-He Sun. I/O acceleration with pattern detection. In ACM Symposium on High-Performance Parallel and Distributed Computing, HPDC 13, New York, NY, June 2013.