Date: **January 9, 2013**	**SOLUTION ARCHITECTURE - IOD** **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O**

SOLUTION ARCHITECTURE - IOD 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

Table of Contents

1. Revision History

Date	Revision	Author
2012-12-13	1.0 Draft for Review	John Bent, EMC Corporation
2012-12-21	1.01 Revised based on Stakeholder feedback.	John Bent, EMC Corporation
2013-01-09	1.02 Added section numbering to TOC	John Bent, EMC Corporation

2. Introduction

Management and efficient utilization of the precious and relatively scarce burst buffer layer is key to achieving the performance required from an exascale storage stack. To accomplish this, POSIX semantics must be discarded in favor of a new storage interface more attuned to extreme scale parallel IO. Our burst buffer management layer, the IO Dispatcher (IOD), exports a storage interface designed for _concurrent,_ _parallel_, _transactional_ access to multiple _versions_ of shared _structured_ _objects_. Specifically, we introduce into our storage stack, abstractions for the following storage features not present in the POSIX API: concurrency, versioning, transactions, and structured objects. Additionally, the IOD introduces _containers_, which logically group objects and allow transactions across them, and highly efficient parallel key-value stores, which evolve traditional filesystem extensible attributes into an abstraction suitable for exascale IO. Finally, the interface to the IOD will be fully asynchronous to allow parallel, concurrent access to shared objects without requiring any synchrony within parallel applications.

Since our burst buffer storage media will be attached to our IO nodes (ION's), we will leverage the fast network interconnect between the IONs. When the IONs are allocated as part of a compute job, IODs will be instantiated on them and provided MPI communication handles by which they can communicate. It is expected that this communication can help the IODs to coordinate data streams for several purposes: one, possible stream aggregation across IODs as data is written from the application layer through the HDF library, two, to coordinate data migration between the IONs and the DAOS layer, and three, to coordinate data shuffle in order to effect the semantic reshaping of data sharding across the IONs. These will be described below.

3. Outside of current scope

There are a few important limitations of the IOD layer that should be noted before delving into deepening scope architectural descriptions. First, the IOD layer will not be making any automatic storage management decisions related to data migration. This means that if the IOD is not explicitly told to migrate data, it may return out of space errors (ENOSPACE) to the higher layer.

However, it will certainly provide mechanisms by which higher layers can explicitly manage the space. In the future, we will consider automated space management within the IOD with only policy directives, and hints, provided by the upper layer.

Second, it will not be possible to store a version of an object, which exceeds the size of the burst buffers. Again, future versions of the IOD may allow this by implementing partial version migration to the DAOS layer but that is currently outside of our scope.

Third, transactions and versions successfully committed to the IOD that are stored within the burst buffers may not be available following a failure to an ION. We are not doing any sort of parity or replica across IONs. Full data protection is only available when data has been successfully migrated to DAOS within a successful DAOS transaction. The IOD will provide mechanisms by which the higher layer can effect the migration as well as mechanisms to query which versions and transactions are committed on the IONs and which are committed on DAOS. Further, there will be no long-term data retention on the IONs. The IONs are allocated as part of the job and when the job terminates, the IONs are cleaned.

4. Solution Requirements

All requirements described herein are mandatory unless specified otherwise.

4.1. HDF5/IOD interface

The IOD must present to HDF5 an interface supporting the abstractions described above; namely, containers, structured objects, key-value stores, transactions, and versioning. Substantial progress has been made already as evidenced within separate documents defining the IOD API, and sample HDF workflows containing pseudo-code illustrating how the IOD API will be used for constructing H5Files from IOD containers and IOD objects as well as operating on them using asynchronous routines within transactional boundaries.

4.1.1. HDF5/IOD interface testing

To demonstrate the completion of the HDF5/IOD interface, we will run an HDF application and show how it exercises the full IOD API to create H5Files using an IOD container for each H5File and IOD objects and key-value stores for all the datasets, groups, datatypes, attributes and links and then that it can successfully write to them and then read the expected data. Testing may proceed incrementally. For example, early testing may demonstrate functionality of the interface with a version of the IOD that supports objects but may not yet support transactions and tagged versioning.

4.2. Object storage

Diverging from traditional POSIX semantics, the IOD API will replace the notion of files within a directory with the abstraction of objects within a container. One of the key benefits with this approach is a more scalable namespace. Indeed, as we have learned so far with our example HDF-IOD workflow, many of the objects created by HDF to comprise an H5File do not require semantically meaningful names at all. Rather, their "names" can be object identifiers generated by the IOD. In turn, the IOD can get object identifiers from DAOS thereby enabling a highly scalable namespace as costly lookups from user-defined name into storage identifier can be avoided.

In terms of how bytes within logical objects will be physically stored within the physical media, IOD will have to manage two different types of mappings: one, mapping logical data into physical locations striped across burst buffers, and two, mapping logical data into physical locations striped across DAOS shards.

At this point, it is not yet clear what the storage interface to the burst buffers will be. For the time being, we will assume a POSIX interface although there was early discussion that a key-value interface might provide higher performance. In any event, the IOD interaction with the storage media will happen within the PLFS code. Modeled after the MPI-IO ROMIO and ADIO abstractions is a similar physical storage abstraction within PLFS. This storage abstraction should allow the IOD to seamlessly store bytes into burst buffers regardless of the burst buffer storage interface.

The more difficult challenge however is the organization of data. Following the architecture of the PLFS foundations within IOD, the initial organization of data as it is written by the HDF layer will be a stream of bytes per writer augmented by a secondary, smaller stream of PLFS metadata which describes how to reconstruct that stream back into the HDF logical object. As the project progresses however, we will certainly consider doing simple MPI-IO style collective aggregation of write streams. This will be examined as we learn more about the performance characteristics of the system and whether an aggregation of streams would be beneficial. If so, the aggregation might happen either across CN's before being shipped to IOD's or it might happen within each IOD. Towards the end of the project, should our discoveries indicated that it will be desirable, we can consider aggregation across IODs.

As said above, each data stream from HDF written into a shared IOD object will be physically stored without modification as a stream of data into a burst buffer. This will use mostly existing PLFS code to store these data streams into the burst buffers. Although the layout code within PLFS will be mostly unchanged in order to support burst buffer storage, other aspects of PLFS will be heavily changed in order to support containers, transactions, and versioning as will be discussed below.

Storage into DAOS however will require heavy modification of the PLFS code. Completed transactions and tagged versions within the burst buffers can be migrated, as directed by a higher layer, into the DAOS layer. Although this discussion will focus on the initial creation and migration of transactions and tagged versions into DAOS, the same data organization will of course be used in order to retrieve data.

A note about terminology in the following paragraph; the reader must please forgive us for we have badly overloaded the term container. The following paragraph discusses three containers: the first is the PLFS container which current PLFS uses to build a logical PLFS file. The second is an IOD container which allows a logical grouping of IOD objects. The third is a DAOS container which allows a logical grouping of DAOS shards.

The DAOS storage interface is a complete, exascale required, departure from the POSIX API to which PLFS was developed. A version of IOD object residing across burst buffers will be stored, as described above, similarly to how PLFS stores files today: within a set of log-structured data streams alongside a smaller set of PLFS metadata used to map the log-structured data back into a coherent, ordered logical file. To migrate these data streams into DAOS will require that they be stored as a set of DAOS objects within a set of DAOS shards within a DAOS container. Further complicating matters is that the IOD must support transactions across multiple IOD objects. PLFS traditionally has created a PLFS container to store the data logs and PLFS metadata comprising a logical PLFS file. PLFS containers are implemented as POSIX directories. The approximation within DAOS most closely similar to a POSIX directories is a DAOS container which seems a natural fit for a PLFS container. However, this will not work as DAOS does not currently allow transactions across multiple DAOS containers. Therefore if the IOD uses a separate DAOS containers to store each IOD object, then IOD cannot use DAOS transactions to implement transactions across a set of IOD objects.

Therefore, PLFS, which relied on POSIX to group data streams, must now implement its own groupings such that multiple IOD objects can co-exist within a single DAOS container.

Fortunately, the DAOS shard abstraction provides a rich functionality which can be used for this purpose. DAOS containers group DAOS shards. DAOS shards map to DAOS storage targets. DAOS shards group DAOS objects but more interestingly they provide an almost infinite allocation space within a two-dimensional extent map. Importantly, the DAOS shard performs extremely efficient sparse space management. The almost infinite virtual space within a DAOS shard costs nothing when it is not filled. Much like the compiler uses virtual memory, so too can IOD use the virtual space with DAOS shards. As the compiler builds a heap from one end and a stack from the other both working towards the middle, so too can the IOD build metadata from one side of the shard and data from the other.

Specifically, we plan for IOD to build an IOD container using a DAOS container in the following way. An IOD container will map directly to a DAOS container and will store all IOD objects into the container shards. The list of IOD object IDS within an IOD container will spread across all of the shards' 0th columns. Each IOD object ID will itself specify a {shard,column} pair. That column will then itself hold a list of {shard,column} pairs identifying where the metadata and data contents of that IOD object are stored. To query a list of IOD objects within an IOD container, first the IOD must know how many DAOS shards there are within this container. It can either ask the container or it can record that itself at {0,0}.

4.2.1. Object storage testing

Object storage testing will extend upon the HDF/IOD interface testing to further test both the cases in which objects are sharded across burst buffers as well as the case in which objects are sharded across DAOS shards.

4.3. Data migration

The IOD will be responsible for providing mechanisms by which upper layers can request that data be migrated between the IONs and the DAOS layer. Additionally, the migration routines can, for structured objects, specify a semantically meaningful sharded organization. The current data migration mechanism shows within the IOD API in the iod_obj_set_layout function. Querying a data sharding within an object is done using the complementary iod_obj_get_layout function.

Data migration will also be responsible for creating storage friendly streams of data. This means that the sharding organization on the burst buffers may not be directly copied into DAOS shards but might be reorganized. This reorganization might accomplish several things. One, it might enable better performance for subsequent analysis by sharding data such that analysis tasks can map to shards thereby enabling node local data access. Two, it might enable better performance by doing IO's in block sizes and alignments that maximize performance from the storage media, which may require different block sizes and alignments for burst buffers and for DAOS. Three, a reorganization can be used to reduce the amount of IOD metadata that describes what data is where. As the data is migrated into DAOS, the data migration will be careful to respect the concurrency characteristics of DAOS; in particular, that best performance will be achieved with no more than a single writer to any one DAOS shard column (i.e. N-N writes to DAOS objects).

In terms of the IOD metadata reduction, imagine that HDF has stored a 3-dimensional array using a lot of small unaligned writes. Since IOD will use PLFS, this will result in a large number of PLFS index entries. When IOD then migrates the structure into DAOS, it can reorganize into a sharding that can be very compactly described (e.g. 3-dimensions, vertically sliced, round robined across the DAOS shards at a particular offset within the DAOS shards). Note that the LANL-EMC demo as shown most recently at SC12 already has a separate process for migrating data between burst buffers and scratch storage. This data migration for this Fast Forward project will resemble that data migration except it will be much more sophisticated. That data migration did a 1-1 copy of each file within a PLFS directory and then did some simple manipulation of the PLFS pointers, which indicate what data, is where. This data migration will be much more akin to a parallel MPI-IO job in that multiple data migration processes across the ION's will cooperate to read, shuffle, aggregate, and write data. This is very much akin to MPI-IO collective IO and we will certainly try to leverage that code as much as possible. Either by directly using that code or by borrowing key ideas.

4.3.1. Data migration testing

To demonstrate the correctness of data migration, we will extend the current 'plfs_query' tool to ensure that it provides the full information about object layout sufficiently descriptive to assure that data has been migrated as required. We will build a test routine that asks for existing objects to be migrated back and forth and use the 'plfs_query' tool before and after each migration to ensure that the migration worked as expected. Since the IOD API is almost entirely asynchronous, the test routine will have to request a data migration, and then wait for its completion before checking status with the 'plfs_query' tool.

4.4. Group transactions and versioned objects

The IOD layer will provide mechanisms by which groups of processes can use the same transaction identifier to asynchronously make modifications across a set of IOD objects while preserving the data integrity and persistent data views of other processes, which may be reading earlier versions of the objects. The one constraint is that the set of objects within a transaction must exist within the same single container.

When an object is opened for read, it creates a view of the persistent object at the time that it was opened. That view will show only fully completed transactions and can be referred to as the most recent consistent view, which is conceptually identical to Highest Consistent Epoch in DAOS. Any uncommitted data belonging to pending transactions at the time of the read open will not appear within the view. This is easily enabled within the IOD layer due to the copy-on-write semantics of PLFS. Distributed processes that want to share a read-view of an object can coordinate to share a version tag that they provide when they open the object for reading.

As multiple transactions complete, older transactions may be flattened into new transactions. This reduces storage usage as well as preventing arbitrary growth of metadata however the cost is that arbitrary time-travel to view the objects at any and all transaction identifiers will not be possible. However, arbitrary time-travel, when explicitly requested by a higher layer, will be possible by an IOD mechanism to allow giving version tags to containers. Those version tags will become part of the namespace and will not be flattened. Note that the transactions and version tags effectively operate on the container but many of the operations are applied on individual, or sets of, objects.

Views of transactions are similar to z-buffers in graphics in which there are multi-dimensional layers of polygons. A view into the z-buffer shows whichever polygons are visible. Visible regions of polygons are those that are not blocked by a region of other polygons at higher layers. Transactions similarly layer on top of each other. To view an older transaction is analogous to the view of z-buffers if the upper layers are removed. This is also similar to layers in graphics tools like gimp, xfig, and MS PowerPoint in which objects can be sent forward and backward. Please also see the DAOS VOSD scope architecture, which also explains this same idea of transaction layers and flattening.

Group transactions in the IOD layer will operate differently when the data is in burst buffers and when the data is migrated to DAOS. The IOD layer will not do any flattening. Instead it will just do copy on write for all incoming writes. It will use existing PLFS type index entries to mark what data is where and will augment the index entries with a transaction. These group transactions can be completely asynchronous and all processes cooperating in a group transaction can show up at any time and initiate a transaction. They can also show up at any time and complete it; each must complete after they themselves have initiated but they can complete before others in the group have even started. The restriction is that each knows the size of the group and specifies this upon initiation. There is also support for a group of transactions to elect a leader and have just that leader initiate and complete. This then requires that no other processes attempt writes before the leader has initiated the transaction.

The IOD layer will track active, aborted, and committed transactions by creating a "container leader" on one of the ION's. Whenever a process asks any of the IODs to initiate a transaction or to request a new transaction ID, that IOD will forward that request to the "container leader" who will be responsible for maintaining the list of transactions for the container. Using MPI and the fast interconnect connecting the ION's and a small cluster of two thousand or so ION's, this solution should be sufficiently scalable for this project.

As data streams into the burst buffers from the applications, it will stream into PLFS style logs with metadata identifying which data belongs to which transactions. When transactions are aborted, the IOD's may, but probably will not, do garbage collection on the aborted transactions. Rather, when data is migrated into DAOS, the IOD's will merely not replay any IOs belonging to aborted transactions.

Notice that the tricky parts of transactions and versions and overwriting will be handled in only one place in our stack: in the VOSDs within DAOS. This means that the IOD can use copy-on-write semantics in the burst buffers and never flatten nor garbage collect in the burst buffers. Rather it will merely remove entire data streams from the burst buffers once they, and all the valid transactions within them, have been replayed onto DAOS. Note, however, that the replay will not necessarily be a 1-1 replay but may use complicated data reshuffling as described above in the data migration section.

Container level snapshots are required but efficient copy-on-write versioning at an IOD object granularity is optional.

4.4.1. Group transactions and versioned objects testing

To ensure the correctness of our group transactions implementation, we will open an object and write several committed transactions to it with groups of asynchronous processes. Some of the transactions will be given version tags as well. Each transaction will write a different set of distinct data. We will also have several uncommitted transactions open with uncommitted writes pending in each. We will then read from the different version tags and ensure we get the expected data. We will finally open for read at the most recent consistent view of the object and ensure that the data does not reflect any of the writes within uncommitted transactions. The most recent consistent view of the object is conceptually identical to the Highest Consistent Epoch (HCE) in DAOS.

4.5. Structural awareness, KV stores, and data blobs

Within an IOD container, there may exist three different types of IOD objects: key-value stores, data blobs, and array objects. A key-value store is simply an interface that looks just like existing key-value stores but adds transactions and version tags. Additionally, the IOD implementation may choose to shard key-value stores when hints from an upper layer suggest that the key-value store may be large or performance critical. A data blob is simply an unstructured stream of bytes: identical in appearance to a POSIX file except that it will also support transactions and version tags.

However, array objects will have structural awareness. The upper layer will define the dimensionality of some objects, which can be then used to store multi-dimensional objects such as meshes and hypercubes. Additionally, one dimension may be eligible for growth. Transactions and version tags will of course work on these array objects. Writes and reads to and from structured objects will not operate on traditional file offsets but rather using vertices within the multi-dimensional object. In the current IOD draft API, these calls are iod_obj_write_structure and iod_obj_read_structure respectively.

The structural awareness and the ability to control and query sharding using semantic descriptions of multi-dimensional objects provide several key advantages both to higher-layers and to the IOD. One, upper layers can control and query object sharding using compact semantic descriptions instead of very large lists of offsets. Two, upper layers can remove from themselves the burden of remembering the large amount of metadata required to map between semantic dimensionality and offset lists. Three, the IOD may itself be able to avoid this burden of large metadata by using semantic descriptions of dimensionality to create compact patterns describing mappings between semantic dimensions and physical locations both within burst buffers as well as within DAOS shards. Fourth, these compact descriptions of semantic shardings can more easily be shared between computational jobs and the visualization jobs that work in concord. These compact patterns have been research quite a bit recently including Jun He's paper at this year's PDSW.

We will refer to the ability of higher layers to initiate highly complicated data reshuffling as semantic resharding. This semantic resharding will enable data-locality aware co-processed analysis programs to co-locate analysis tasks with data shards both in the case when they are sharded across burst buffers as well as the case in which they are sharded across DAOS shards.

4.5.1. Structural awareness, KV stores, and data blobs testing

To test these different types of objects, we will just ensure that each passes integrity tests for writes and reads since each of the three has different write/read routines. This testing will complement and be complemented by the testing of transactions and tagged versions. Additionally, we will create sets of these objects and use iod_container_list_objects to ensure that each object is of the expected type.

4.6. Support for analysis and reads

Analysis and reads will be supported through the IOD layer. Semantic resharding can shard data both across burst buffers and across DAOS shards in a manner conducive to subsequent analysis in which data-locality is important. Data consistent in the face of concurrent writes to shared objects will be allowed via transactions and tagged versions.

4.6.1. Support for analysis and reads testing

We will devise additional unit tests for semantic resharding in addition to all of the testing described above for object types, object IO, and transactions and tagged versions. These unit tests will use the iod_obj_set_layout to affect a data sharding and then use iod_obj_get_layout to test whether the sharding was done accurately. Additionally, lower level tools such as plfs_query can double-check to help find any shared bugs in iod_obj_set_layout and iod_obj_get_layout. Finally, performance testing will also show whether data that should be local actually is.

4.6.2. Data reorganization and data reorganization testing

Please refer to the description of semantic resharding in the KV Stores subsection above.

4.7. Multi-format replicas

A multi-format replica is the idea that semantic resharding might create multiple replicated object shardings. Interestingly, none of the replicated shards may be identical to any other. But rather, the entire multi-dimensional object can be recreated from different sets of shards. For example, a three dimensional cube might be fully replicated across burst buffers by being both sharded as vertical slices as well as also being sharded as horizontal slices.

To allow this, PLFS indexing will have to be augmented to allow replicated data. We may implement this by requiring that multi-format replicas are given different version tags. As new transactions are committed and new version tags are applied, multi-format replicas may or may not be guaranteed to remain visible although they can always be recreated upon demand.

4.7.1. Multi-format replicas testing

To test this, we will create multi-format replicas and use iod_obj_get_layout, plfs_query, and performance measurements to check that the multi-format replicas have been sharded as directed by the upper layer.

4.8. End-to-end Data Integrity

The IOD will cooperate with the function shipper to achieve end-to-end data integrity between the application and the storage media. When HDF writes data to IOD, the function shipper will send the data buffer plus a checksum. IOD will then store the data buffer and the checksum into its data stream. An alternative would be to store the checksum into the metadata stream but this creates too much metadata and prevents pattern-based compression of the metadata. When data is read, if the read request matches an existing write buffer (i.e. the read is requesting an exact range of data that was previously written), then IOD will give that buffer and its checksum to the function shipper. If the read request does not match an existing write buffer, then the IOD will have to read every write buffer from which it must pull data and copy that data into the read buffer. It will then checksum the read buffer, and then check the checksum of the write buffers. If everything is correct, it will return the buffer and the checksum to the function shipper. When the function shipper returns the buffer to the reader, it first checks the data integrity. Since there is no memory copy between the function shipper and the reader, this provides end-to-end data integrity.

4.8.1. End-to-end Data Integrity testing

We will flip bits in the data buffers and in the checksums by direct manipulation of the backend storage (i.e. without going through IOD). We will then attempt to read the data and ensure that the data corruption is detected.

5. Acceptance Criteria

Demonstrate success for all testing as described above.