Date: **June 4, 2014**	**Versioning Object Storage Device (VOSD) Design Document** **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O**

Versioning Object Storage Device (VOSD) 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

Table of Contents

Revision History

Date	Revision	Author
Feb. 21, 2013	1.0	Paul Nowoczynski & Johann Lombardi
Mar. 03, 2013	1.1	Paul Nowoczynski
Mar. 29, 2013	1.21 (Comments and Responses)	Paul Nowoczynski
Jun. 04, 2014	1.3	Liang Zhen

Introduction

This design documents details the integration of the Versioning Object Store into the existing Lustre Object Storage device.

Definitions

- **VOSD** – "Versioning Object Storage Device"
- **HCE** – Highest Committed Epoch.
- **HCE+1** – The next epoch on the commit horizon.
- **> HCE + 1** – Epochs which are further on the commit horizon. Writes to such epochs are generally directed into the Intent Log.
- **OI** – Object Index. An object index is a stateful Lustre OSD structure which enables the mapping of a Lustre based identifier to a file or directory in the underlying filesystem (such as ldiskfs or zfs).
- **Versioning Intent Log (VIL)** – On-disk log to cache pending transactional data from uncommitted epoch higher than HCE+1.

Changes from Solution Architecture

During this last quarter it was concluded that writes into a VOSD on behalf of epoch *HCE+1* may be placed directly into the shard dataset without requiring mediation by the intent log. This technique does not violate VOSD semantics and hence, represents a significant breakthrough by providing means for avoiding intent log replay overhead in some situations.

Another method for addressing intent log and snapshotting overhead in VOSD was devised in the previous quarter. The method specifically addresses situations where applications may pathologically issue epoch commits such that the VOSD intent log flattening and snapshot overhead become dominant. Working in conjunction with the metadata server, the VOSD may be instructed to 'flush' rather than 'commit' should the MDS determine that the commit frequency on the container is too high. Unlike a 'commit', which requires synchronous intent log flattening and a shard snapshot, 'flush' is a suggestive instruction which compels the VOSD to begin flattening an intents stored on behalf of the given epoch. This method is described in a later section called *DAOS Transaction Aggregation*.

Overview

To a large extent, VOSD functionality may be achieved by exposing functionality within existing copy-on-write filesystems. More specifically, the snapshot and rollback provide a basis for the VOSD framework in such a way that does not require DAOS to completely reinvent, or develop from scratch, a capable CoW filesystem. During this last development quarter the VOSD team decided to move forward with the ZFS filesystem for providing the basis of VOSD over btrfs. The reasoning behind this decision lied with

ZFS's proven stability and its existing integration with Lustre. Btrfs was compelling due to its open-source backing and GPL licensing but ultimately proved too risky given the scope and time frame of the Fast Forward project. In the long term btrfs may prove to be the dominant CoW filesystem. Should this be the case in the future, we expect that the techniques used to implement a ZFS-based VOSD will be largely commutable to btrfs.

Starting with the current ZFS-based Lustre OSD provides a strong foundation for this work. For instance, the current ZFS Lustre OSD already fully supports I/O operations into ZFS volumes as well as Lustre specific functionality such as object index support. Despite the fact that ZFS on Lustre has proven to be successful, the means by which Lustre makes use of ZFS are relatively simplistic when considering the overall feature set of ZFS. The successful implementation of VOSD will require these functionalities to be exposed within the Lustre OSD framework in a manner which supports DAOS operation.

Establishing a VOSD within the context of ZFS and Lustre requires a moderate degree of modifications to both. The Fast Forward design team is going to great lengths to find a reasonable balance between the architectural integrity of VOSD and integration into the existing implementations (of ZFS and Lustre).

Characterization of Lustre Modifications

The implementation of VOSD improves upon the Lustre OSD by advancing the internal filesystem framework such that fundamental ZFS features, such as snapshot and rollback, may be exposed to DAOS. This work extends the current ZFS OSD implementation by allowing for more native ZFS capability to be exposed. Currently, the ZFS OSD implements the same API as the other OSD filesystem, ldiskfs (which is non-CoW).

Conceptually shard is virtual target of container, so the initial idea is mapping DAOS shard to Lustre OST device, however, it will require full stack changes and need VOSD to support the on-demand instantiation of devices, which is going to be too much efforts for a research project.

To simplify implementation, a less structurally aligned way is chosen in this design: shard is represented by special object of VOSD, it is acting as multiplexer of IO requests to DAOS objects, which are ZFS dnodes and invisible to upper layer server stack.

Unlike regular OSD object which associates with dnode, each VOSD shard has a dedicated ZFS dataset, it also has private Object Index (OI) to map DAOS object ID to ZFS dnode number.

To handle DAOS specific IO requests, VOSD shard has two sets of methods:

- IO functions

  They are similar with regular OSD object IO functions except a major difference, these functions require extra parameters as three-dimensional address of bytes in shard: epoch, ID and offset. Epoch is identifier of transaction or snapshot, ID is 64-bit DAOS object number, offset is IO start address for DAOS object.

- Epoch functions

These are shard-only methods that can handle epoch requests like commit, flush, slip, abort, stat and query. More details can be found in later section of this document.

Modifications for Supporting Intent Logging

Supporting the intent log functionality requires substantial engineering of ZFS and subsequent integration into VOSD. For instance, intent log writes must be seamlessly redirected from the upper layers of the Lustre OSD into their respective intent log objects. Furthermore, writes directed to the IL must be translated from pure Lustre writes to the intent log format.

To avoid extra overhead of copying all data from VIL to target object while replaying log, the most important feature of VIL is Zero-Copy (ZC) write, full block writes should only be performed once instead of writing data into log first and copying them later. In ZC write mode, when full block writes arrive at VIL, new blocks are allocated for VIL and block pointers are store in VIL log record. When VOSD starts to replay log records, it should be able to migrate these blocks to destination object without any data copy.

Specification

DAOS Shards

Within a VOSD ZFS pool, a shard is represented by two types of ZFS dataset. These are described immediately below.

1. Staging Dataset (HCE+1)

Staging dataset is where writes for epoch HCE+1 land, it is a staging area where the next stable snapshot is groomed.

The staging dataset accumulates updates to HCE+1 through two methods.

The first method is through direct write. By mandating the snapshot of HCE on commit, VOSD may allow for writes to HCE+1 to be applied directly into the staging dataset. As described earlier in this document, this is an important technique for allowing the intent log to be bypassed in many circumstances.

The second method the staging dataset accumulates updates is through intent log flattening. It should be noted that DAOS does not guarantee ordering of writes within a single epoch. Therefore, the staging data set may be prepared through both methods - direct write and intent log flattening - simultaneously.

The staging dataset will be snapshotted and become the new stable version on successful commit.

To support Zero-Copy (ZC) write with less engineering to ZFS, block pointer reassignment requires data block belonging to the same dataset, which means VILs have to stay in staging dataset as well. The major downside of this approach is, all VILs for future epochs will be snapshotted on commit, they will consume extra space because there could be large amount of data in VILs, this issue may be fixed in follow-on project in the future.

Staging dataset also has ZAP (ZFS Attribute Processor) to store entries for all available epochs of shard, including both stable versions that are identifiers of snapshots, and uncommitted versions that are identifiers for uncommitted VILs.

To route IO request to ZFS dnode within shard, DAOS object ID has to be mapped to dnode number, so each staging dataset should have private Object Index (OI) to handle this conversion.

OI will be snapshotted as part of snapshot on commit, so reader can find corresponding dnode as well by DAOS object ID.

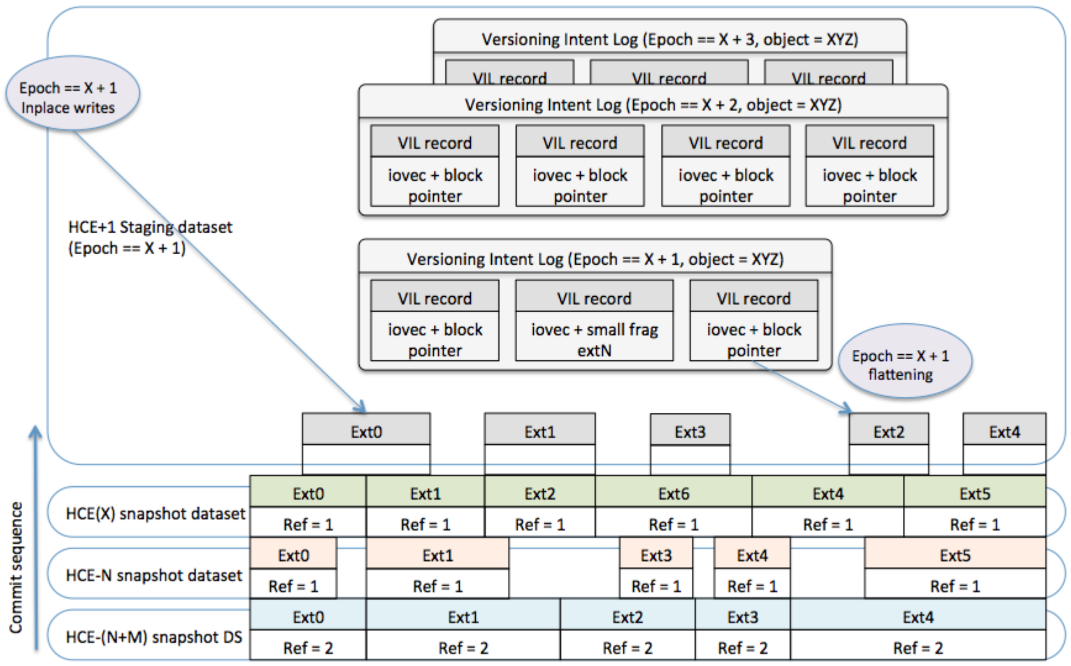

Figure 1 - Shard with Intent Logs and Datasets

There is only one instance of staging dataset for each shard.

2. Snapshot dataset

The second dataset type of shard is snapshot which is stable version of the shard. Snapshot will not exist for newly created or otherwise unused shards. By definition, per the VOSD specification, snapshots of HCE and earlier epochs are read-only.

A non-empty shard should at least have snapshot for HCE, a shard may have multiple snapshots for old epochs as long as readers have reference on them. These old snapshots can be recycled on demand when application calls 'slip' for corresponding epoch, or recycled on shard close, which also implies epoch 'slip'.

Routing DAOS Requests to the Appropriate Shard Instance

As described in Figure 1, the Lustre storage stack will not be 'shard-aware', shard is a special three-dimensional object to upper layer, within shard, DAOS object is mapped to underlying ZFS dnode via epoch number and object ID.

As previously mentioned, Object Index (OI) exists in both staging dataset and snapshot, this design alleviates the need for the current Lustre OI layer to be aware of the epoch number thus insulating these parts of the stack from disruptive API changes. When the VOSD shard receives IO requests for HCE+1 or lower epochs, it first finds snapshot or staging dataset according to epoch number, then routes request to underlying ZFS dnode by searching object ID in OI.

.

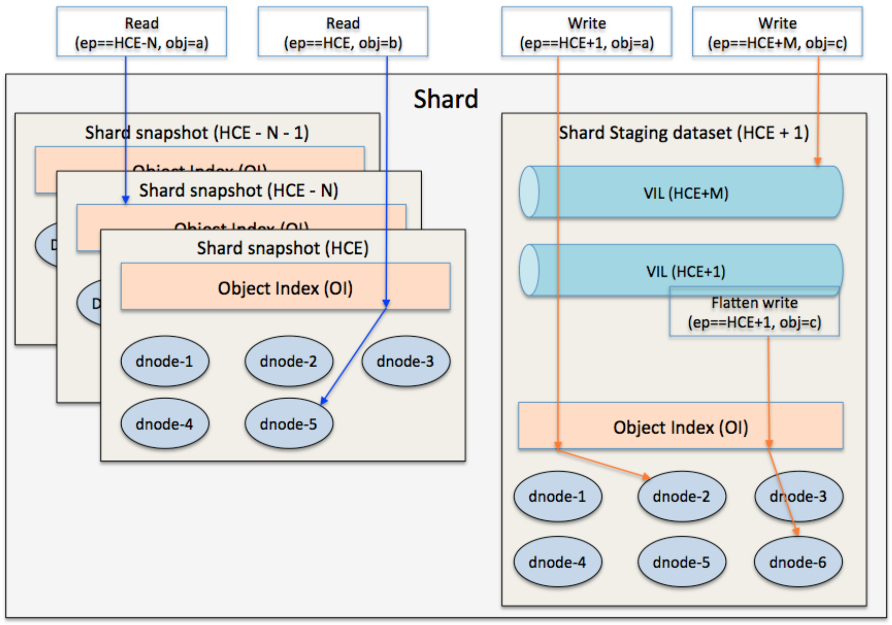

Figure 2 - Routing DAOS requests to shard

Routing to Intent Logs

VOSD shard object maintains a hash table for VILs which is indexed by epoch number. All existent VILs of shard will be loaded from ZAP and inserted in hash table on the first access of shard.

When shard receives IO requests for epoch higher than HCE+1, it needs to find corresponding VIL in hash or create VIL for this epoch. After that, it calls VIL IO routines to store both data and IO descriptor as log record. All log records will be replayed on commit.

The process of replaying VIL records is called as 'flattening'. While flattening VIL, shard needs to search logged object ID in OI of staging dataset to find destination dnode, and apply changes to dnode.

Versioning Intent Log (VIL)

A VIL is a specialized structure which is purposed for buffering writes into epochs beyond the commit horizon. Specifically, this means a write to any epoch > HCE + 1 must be staged into an intent log. As described in previous section, intent logs are manifested on a per-shard, per-epoch basis, they reside in staging dataset to simply ZFS engineering for zero-copy write.

Each VIL is composed of a ZIL (ZFS Intent Log) and a special ZFS dnode. ZIL is responsible for recording descriptors of IO requests, the special dnode is the ZFS object to store full block writes for zero-copy.

The components of the VIL write record will be as follows:

```
Typedef struct {

        lr_t            lr_common;      /* common portion of log record */

        uint64_t        lr_foid;        /* file object to write */

        uint64_t        lr_offset;      /* offset to write to */

        uint64_t        lr_length;      /* user data length to write */

        uint64_t        lr_blkoff;      /* block offset in file object */

        blkptr_t        lr_blkptr;      /* spa block pointer for replay zero-copy write */

} lr_write_t;
```

When full-block writes arrive at a VIL, new blocks are appended to the special dnode, at the meanwhile, block pointers are saved in lr_write_t::lr_blk_ptr as part of log record. On flattening, VOSD will relocate the block pointers of these blocks into the corresponding DAOS object as it exists in the staging dataset. This capability is a deviation from the ZFS ZIL, which copies all logged data contents into its respective dataset. One advantage of this method is that logged blocks may be easily reclaimed in the event of a failure or rollback. Additionally, this method guarantees that the ZFS I/O pipeline has manicured the blocks in preparation for their association with a file. This means that the blocks' checksums may be reused without need for recalculation.

Intent Log Flattening

Each entry self describes its type and size in manner which is consistent with most filesystem journal implementations. For DAOS writes smaller than a single ZFS block, the contents of the write are appended to the log entry and the intent log data length structure member is set to mark the length of the write. This length value also allows the next intent log entry to be easily located amidst variable length data. In the case of full block writes, the intent log entry stores a copy of the ZFS block pointer information so that the block may be incorporated into the destination dataset with zero-copy.

Flattening of the intent log is a relatively simple procedure which is prompted by the MDS in preparation for epoch commit. Upon receiving notification from the MDS, VOSD first locates the IL associated with the commit operation and then begins to iterate over the intent log entries encoded in the log[1]. Small writes are issued through the standard dmu_write() interface in a manner similar to ZFS's own intent log (ZIL). While large writes are processed without dmu_write() but rather by block pointer reassignment.

Block pointer reassignment must be cognizant of the ZFS internal structures needed for maintaining garbage collection and space accounting.

Eager Preparation of the Staging Dataset to Reduce Commit Latency

Writes from DAOS clients are directed per their shard FID to their respective VOSD shard. When the epoch number of the incoming write is > HCE+1, the write is directed to its respective intent log. This element of the VOSD design has been in place since the beginning stages of the design.

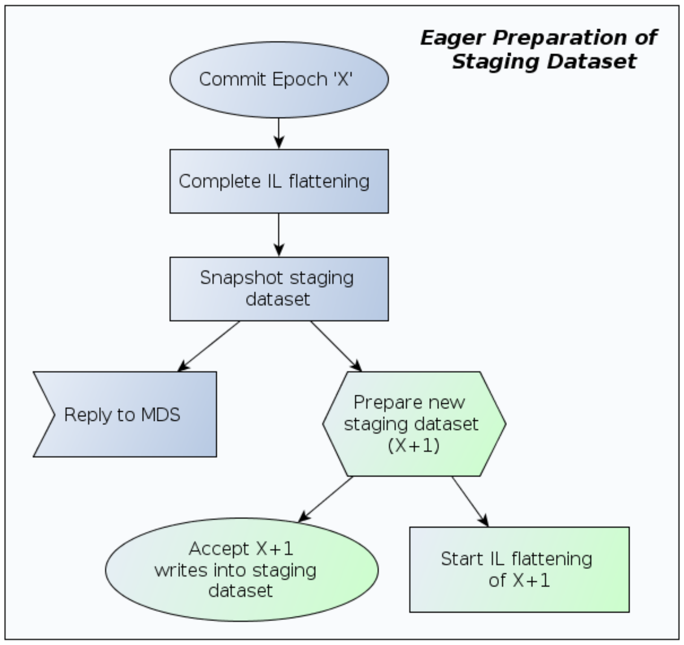
Eager Preparation of Staging Dataset
Commit Epoch 'X'
Complete IL flattening
Snapshot staging dataset
Reply to MDS
Prepare new staging dataset (X+1)
Accept X+1 writes into staging dataset
Start IL flattening of X+1

Figure 3 - Eager Preparation of Staging Dataset and Intent Log Flattening

However, in the case where a set of incoming writes' specified epoch is HCE+1, the deltas may be applied directly to the staging dataset without mediation by the intent log. This optimization is meant to reduce intent log replay for HCE+1 writes.

Figure 2 shows a shard staging dataset and intent log structures. The diagram shows ingest from simultaneous epochs and flattening activity which occurs while the write window to HCE+1 is still open. This is another type of optimization which is meant to

[1] Note that underlying DAOS objects and their respective OI structures are *created-on-write*. The create-on-write construct is carried on within the intent log replay such that intent log replay actions may implicitly create new DAOS objects.

reduce commit latency by ensuring that intent log flattening begins at the earliest possible moment.

The Figure 3 flowchart shows how staging dataset preparations begin immediately following the commit of the previous epoch and facilitate eager flattening. Once epoch X+1 has been promoted to the staging dataset, writes from DAOS clients and intent log replay activities may occur simultaneously within the staging dataset. This is possible due to the transactional specification of DAOS which guarantees that updates within a given epoch are non-conflicting.

Epoch Commit

A commit is executed by an application once all of the writes composing the given epoch have been submitted to DAOS. The commit request is presented to the MDS on behalf of an individual container. The MDS is aware of the container's layout and proceeds to setup a collective operation which encompasses the container's set of shards. Upon receiving a commit request, the VOSD prepares the staging dataset to be snapshotted. At this point, the VOSD may safely assume the application has closed the update window and that the epoch submitted for commit, and all lesser epochs, have quiesced. Before responding with a completion message to the MDS, VOSD must ensure that the any intent logs involved in the commit operation have been flattened. If the commit epoch is > HCE+1, intent logs of epochs falling between HCE+1 and the commit epoch must be flattened sequentially to preserve the cross-epoch ordering semantic.

Once flattening operations have completed, the VOSD snapshots the staging dataset and replies to the MDS. At this point, VOSD cannot rollback committed epoch anymore, which means if any other shard failed to commit, container is stuck in a partial committed epoch. Application should either wait for recovery of failed shard or exclude the failed shard if there is replica for it. After that, user can require MDT to initiate collective operation and commit the same epoch again.

Shard should ignore commit request for already-committed epoch and reply success to MDT, because all epoch operations are idempotent.

Rollback to HCE

If application crashed before completing a transaction and recovered, it can either continue IO and commit transaction after reopening the container if it knows status of transaction, or abort all uncommitted transactions because it does not know whether IOs get lost on crash. For the later case, VOSD needs to rollback shard to HCE which is known to be consistent.

To rollback to HCE, VOSD needs to create a clone of HCE snapshot because HCE is the latest consistent version of staging dataset and it is read-only, after that, VOSD can promote this clone as new staging dataset and remove the original one. As described in earlier section of this document, VILs may be snapshotted because they reside in staging dataset, which means there could be dirty VILs in the new staging dataset after rollback, so VOSD should also remove all these VILs after rollback.

Aborting an uncommitted Epoch

Application can explicitly abort an epoch, all changes in requested epoch and later epochs will be abandoned.

If requested epoch is higher than HCE+1, VOSD can simply delete VILs for this epoch and higher epochs. If specified epoch is HCE+1 and changes may have been applied to staging dataset, then VOSD also needs to rollback to HCE in the exactly same way of previous section.

DAOS Transaction Aggregation

DAOS applications which request frequent commits, by default, may invoke ZFS snapshots at rate which is not germane for high performance I/O to the underlying ZFS pool. This is because the snapshot forces synchronous activity at the ZFS layer. To deal with this phenomenon the DAOS metadata server may convert an application's commit request into VOSD *flush* operation which, in turn, is handled differently by the VOSD. The result of this behavior is that application commit actions may not result in a container snapshot. Hence, an application commit is merely notifying the storage system that the update window for that epoch is closed (since a snapshot is no longer guaranteed). Depending on the decision taken by the metadata server, with consideration to the application's commit frequency, a container snapshot may or may not be created.

When VOSD receives a flush operation, it proceeds as if a normal commit has been instantiated with the exception of the snapshot itself. Intent log flattening for the *flushed* epoch must complete before the VOSD replies to the MDS. Once this flattening has

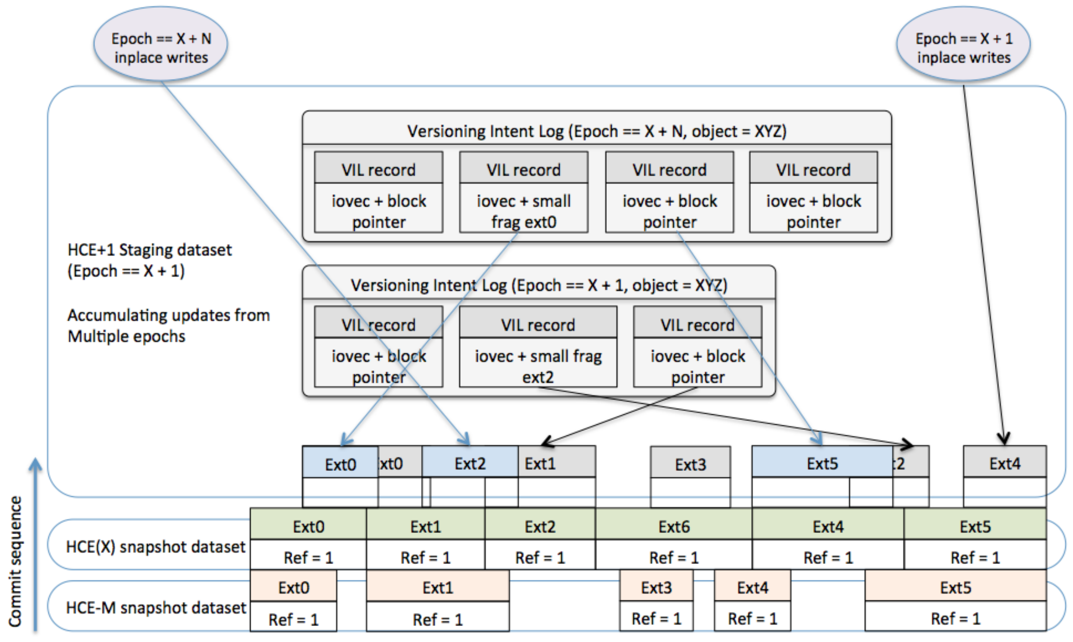

Figure 4 – Accumulation of writes for multiple epochs into the staging dataset

completed the staging dataset, which would have been snapshotted on a typical commit, is ready to accept direct writes from DAOS applications and begin intent log flattening for the following epoch.

The MDS collects the results from the *flush* collective but does not acknowledge the commit operation to the DAOS application. Acknowledgement is delayed until the next snapshot is taken to ensure the application's semantic integrity. The asynchronous nature of the DAOS clients' event queues allow for the applications to progress without needing to block on a typical commit operation. Since commit no longer guarantees a container snapshot, the DAOS application may call daos_epoch_commit() with a 'sync' flag which overrides any MDS decision to convert a commit into a flush, thus restoring some control to the application regarding explicit snapshotting.

While this approach spares the VOSD from snapshotting too frequently, it comes with the caveat that the staging dataset now contains updates from two or more epochs – making it impossible to separate the deltas applied on behalf those epochs.

API and Protocol Additions and Changes

- **`static int osd_dso_shard_flush(const struct lu_env *env, struct dt_object *shard, __u64 epoch)`**

  `osd_dso_shard_flush is issued by client when application calls daos_shard_flush for a shard. This function will wait for pending changes of epochs smaller or equal to 'epoch' to be flushed to storage.`

- **`static int osd_dso_shard_commit(const struct lu_env *env, struct dt_object *shard, __u64 epoch, bool flush_only)`**

  `This call is issued by the MDS on behalf of a DAOS application. Depending on value of 'flush_only', the MDS will decide whether or not to instruct the VOSD to create a snapshot for the shard.`

  `In the case where 'flush_only' is true, VOSD will flatten the appropriate intent log(s) but will not call snaphot. Otherwise, both intent log flattening and snapshotting of the shard will occur.`

- **`static int osd_dso_shard_slip(const struct lu_env *env, struct dt_object *shard, __u64 epoch)`**

  `Release the reading reference count on snapshots of all epochs earlier than 'epoch', if any of these snapshots has zero reference, it will be recycled. This function is called by MDS on behalf of a DAOS application.`

- **`int osd_dso_shard_abort(const struct lu_env *env, struct dt_boejct *dt, __u64 epoch)`**

  `Abandon all changes in 'epoch' and earlier epochs, MDS will issue this call when user wants to abandon all data in uncommitted epoch by calling API daos_epoch_abort.`

- **`int osd_dso_shard_stat(const struct lu_env *env, struct dt_object *dt, __u64 epoch, struct obd_statfs)`**

  `Collect statistics like space usage, number of dnodes etc for a committed epochs or the staging dataset (HCE+1), application can call this function from remote client stack by calling API daos_shard_query.`

- **`int osd_dso_shard_epoch(const struct lu_env *env, struct dt_object *dt, __u64 *epoch_out, enum dt_shard_epoch_op op)`**

  ```
  Collect epoch information like HCE, epoch of the stating dataset.
  When 'op' is DT_SHARD_OP_LOOKUP_STAGING, HCE+1 will be returned
  to 'epoch_out', when 'op' is DT_SHARD_DT_LOOKUP_HCE, HCE will be
  returned 'epoch_out', these are the only options for the time
  being.

  This is a helper function which cannot match to DAOS API, it is
  called when MDT wants to open a shard.
  ```

Open Issues

VILs stay in staging dataset and will be snapshotted, this should be changed if blocks can migrate between different datasets.

Risks & Unknowns

Container Snapshot Frequency

At this point it is unknown what impact a high container snapshot frequency will have on the VOSD. Therefore, we have begun to put in place measures which will allow the snapshot frequency threshold to be configurable parameter.

Implementation Complexity of the Intent Log

Modifying ZFS internals to support zero-copy behavior for the VOSD intent log will require complex modifications to the ZFS layer. It is likely that issues may be found which impede progress in this area. This had come to light after our decision to use ZFS and the subsequent analysis which revealed that the ZFS intent log (ZIL) does not utilize zero-copy operation for full block writes.

Comments and Responses

Question Regarding ZFS Implementation Complexity

***Do we have any estimation as to the level of complexity of changes to ZFS? While this is research, it would be good to have some notion of how required changes within both ZFS and Lustre will be able to make it upstream at some point*. [GMS] -- Page 2, Paragraph 2**

Modifications outside of the Zero-Copy Intent Log implementation are considered attainable and therefore, low risk. The reason for this is that no new ZFS internal APIs are needed to implement the core VOSD functionality. The team is leveraging existing ZFS APIs for managing ZFS datasets to implement checkpointing and rollback functionality.

Routing of Writes Belonging to Epochs >HCE+1

(Note both question are addressed with the answer below)

***Are all writes destined for >HCE+1 always directed to the intent log or are direct "extent writes" supported for >HCE+1?* [GMS] – Page 5, Diagram 3**

***Same question above, all writes to >HCE+1 are directed to the Intent log?* [GMS] – Page 5, Paragraph 4**

At the point of the document where this comment was presented we had described the scenario where direct writes for >HCE+1 were allowed into the staging dataset. To answer this question directly, the answer is 'yes'. The section titled 'DAOS Transaction Aggregation' describes the condition in which direct writes for >HCE+1 are able to occur.

Handling of Conflicting Writes within an Epoch

***I assume there is no issue with writes within the intent log conflicting with writes already applied directly to the staging dataset? That is to say, the last applied write will "win".* [GMS] – Page 8, Paragraph 2**

Correct. The last write *applied* will win (not the last write issued). DAOS assumes that applications are properly managing their own consistency within an epoch. Therefore, if an application issues a conflicting or overlapping write, DAOS makes no guarantees which write will appear in the object.

***This may be better stated that DAOS makes NO guarantees as to ordering of conflicting writes within a single epoch.* [GMS] – Page 8, Paragraph 3**

Noted. Thank you.

Question about DAOS Transaction Aggregation and Snapshot Bypass

***So if the next epoch commit fails, they roll back two versions as HCE+1 was never truly committed and they have moved forward to HCE+2. [GMS]* – Page 9, paragraph 5**

In the case where a 'commit' was changed into a 'flush' a rollback may revert back 2 or more epochs. The number of versions involved is related to the rate of commits the application was performing. This is a trade-off between commit frequency (which affects the entire ZFS pool) and the application's rollback granularity.

***So the application was not acknowledged so they have no guarantees as to the durability of that epoch. Failure of their current epoch could result in rollback to a version two epochs ago. [GMS]* – Page 9, Paragraph 6**

Correct. When a 'commit' mutates to a 'flush' the application has not been guaranteed durability for the epoch which it committed. From a semantic perspective this should be a tenable approach.

Regarding the Implementation of the Zero Copy Intent Log

***What is the long term view for these ZFS modifications? [GMS]* – Page 12, Paragraph 2**

The long term view is to approach this problem once the core VOSD implementation has been sorted out. While this is a critical optimization, it is still an optimization which doesn't affect the core implementation of the VOSD core. At this point the team is highly committed to implementing this feature into ZFS though our current focus has been on the VOSD / Lustre integration. The goal is to have the zero copy intent log patches pushed upstream into the ZFS tree.