whamcloud

Solution Architecture
For
Versioned Object Store

Revision History

Date	Revision	Author
12/2/2012	1	Paul Nowoczynski, DDN
1/06/2013	2	Paul Nowoczynski, DDN

Table of Contents

Introduction

Versioned object store is an advancement beyond current object storage techniques which enables the input and output of per-object byte streams based on sequence value provided by an application. The technique provides the foundation for an exascale filesystem by removing the need for global read and write locking over an object's address space. To accomplish this 'lockless' behavior VOSD assumes a global transactional semantic by which uncommitted epochs are assumed to be incomplete. Byte streams stored on behalf of uncommitted epochs are not exposed for consistent reading. Conversely, versions which have been committed via global transaction are made immutable. By utilizing this versioning technique the system may manage coherency in a more liberal manner – one which does not require explicit serialization when sets of unaligned writes are issued by clients. However, the realization of these features requires the object storage system to support the simultaneous egress and, perhaps more critically, the ingress of versioned byte-streams to or from a given object. Currently no OSD filesystem fully supports the entire feature set needed for versioned objects. The result of this effort would produce the required features by leveraging copy-on-write techniques found in filesystems such as BTRFS and ZFS.

Solution Requirements

All requirements below that are not explicitly marked as optional, are mandatory for successful completion of acceptance criteria.

1. Simultaneous Storage of Multiple Epochs

Versioned Object Stores must be capable of accepting input on behalf of any uncommitted epoch number. Typically transactional-based filesystems such as ZFS maintain changes within transaction groups that are held completely in RAM. However in this case, a VOSD server is an acting member of a global transaction and therefore does not control when a transaction may be completed. This makes the in-memory caching of all pending transactional data impractical since it cannot be guaranteed that a server will have enough memory available to hold all pending transaction state. To deal with this phenomenon, VOSD must be capable of statefully storing arbitrary object extents on behalf any epoch number which is greater than the last committed epoch.

2. Consistent Reads Limited to Committed Epochs

VOSD ensures consistent object reads only from committed epochs. The coherency expectations from the exascale filesystem are different from that of a standard POSIX filesystem. One key deviation from POSIX is that the most recently written extents are not required to be available for reading until the epoch of those writes has been committed. Hence, VOSD must be cognizant of a given object's last committed epoch number so that read requests are not derived from newer, uncommitted epochs.

3. Reads Account for all Previous Epochs

VOSD read operations must account for previous versions when necessary. In the case of multiple committed epochs, VOSD must account for extents from previous epochs which were not overwritten by a subsequent version. Extents from older epochs which have not been overwritten by new epochs must be maintained in the current object view.

4. Epoch Flattening

VOSD enables the flattening of two or more epochs into a single set of extents affiliated with the last committed epoch number. As incoming epochs are committed the pointers to the viable extents are coalesced within a single VOSD object. This is necessary for inline garbage collection and for the efficient lookup of the extents composing the most recent view of the object. Old epochs are cleaned once their readers have departed and unreferenced blocks are reclaimed.

5. Snapshots Derivable from the Last Committed Epoch

VOSD will support the snapshotting of specific epochs and allow reads to be derived from a given snapshot on request. VOSD snapshots will support applications which wish to preserve a given object state. Object snapshotting prevents the flattening of the specified view.

6. High Performance Writes

Writes into VOSD must be capable of achieving high performance regardless of the epoch being written. VOSD must avoid techniques where extents belonging to epoch versions greater than '*current_epoch + 1*' are diverted to special log devices. This is for two reasons:

1. Writing to specialized devices generally requires copies to be made at some point in the future.

2. Given the high bandwidth of large JBOD or RAID systems, a specialized log device will typically have a lower ingestion bandwidth.

Whenever possible VOSD writes must be placed directly into the backing filesystem to leverage the bandwidth of the entire set of HDDs. In addition, flattening activities should not require a copy of bulk data but rather a copy or reassignment of extent pointers.

Use Cases

1. Lockless I/O

VOSD provides the underpinning for lock free I/O through the use of globally transactional epochs. This is considered to be a primary driver for scalability in exascale I/O systems.

2. Simultaneous Writes from Multiple Epochs

Given the multi-tier nature of exascale storage, it is likely that large HPC applications will output several epochs to burst storage prior to any one epoch being committed to the Versioned Object Stores. Therefore it follows that migration of multiple object epochs will be in flight simultaneously. VOSD must be capable of tolerating this behavior in the manner consistent with the solution requirements.

3. Creation of New Object Versions through Partial Overwrite

The flattening feature of VOSD allows for users and applications to create new versions without having to re-output the entire object. Partial updates may be applied to existing objects through the epoch mechanism.

4. Application Driven Object Snapshots

VOSD maintains consistent views for currently accessed versions of a given object. By default, these views are ephemeral and may be garbage collected as newer versions arrive and access references to the oldest version are reclaimed. This feature allows applications to pin a specific object version such that its view will remain intact after the last readers of that version have departed.

Solution Proposal

The Versioned Object Store will be manifested through the modification of one or more copy-on-write filesystems. At this time the two prime candidate filesystem are ZFS and BTRFS. CoW techniques are a natural fit for this problem however neither BTRFS nor ZFS have the entire complement of necessary features to support VOSD in their current form.

The design of VOSD is largely dependent on the use of intent logs for storing object extent metadata belonging to uncommitted epochs. To maintain performance of writes, large extents of uncommitted epochs (UCE) will be stored through the filesystem's primary extent allocation mechanism. Small extents will likely be stored within the intent log.

Figure 1 shows an example versioned object with 3 epochs. The HCE is 0 while the highest UCE is 2. Neither epoch 1 or 2 have been committed but both have extents allocated in the filesystem. Epoch 2 has a small extent which has been stored within the intent log. (Note that the actual implementation may employ one intent log per epoch.)

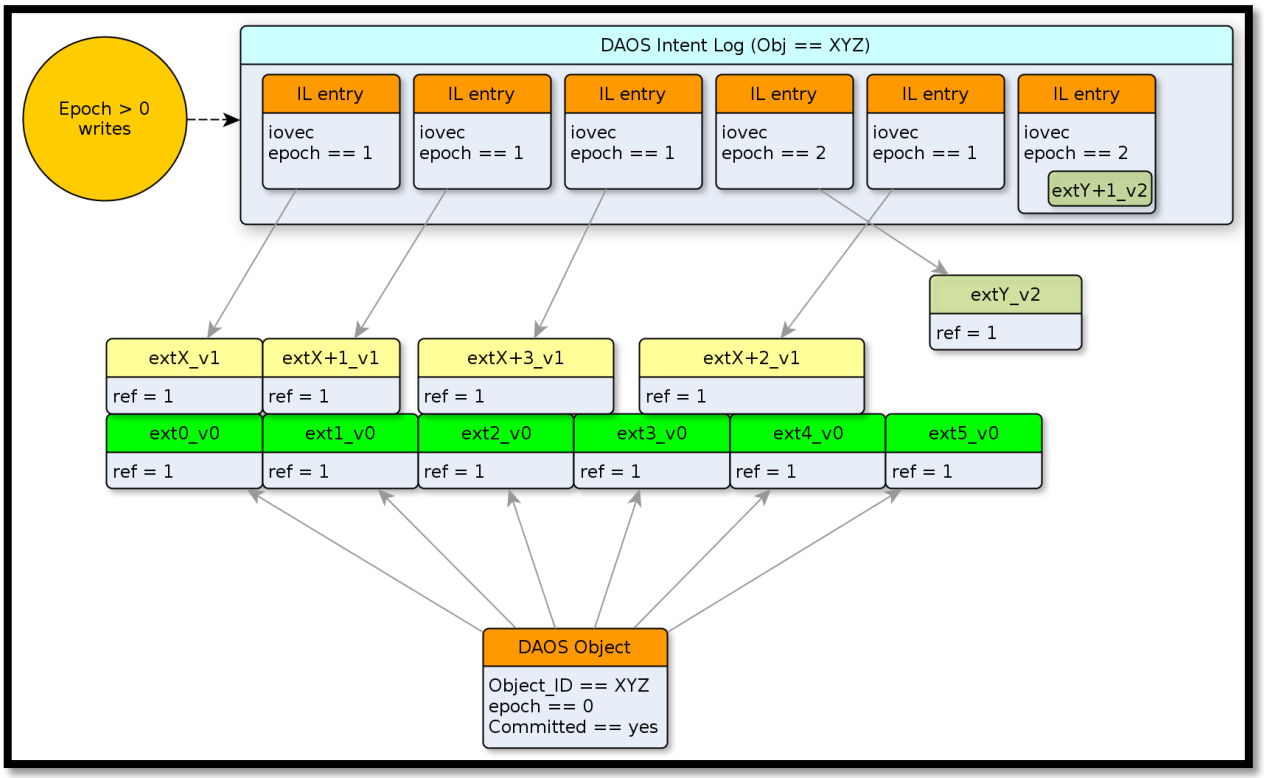

Figure 1 - VOSD Intent Log

Such a strategy requires an extent assignment mechanism which allows extent pointers to be affiliated with intent log entries as opposed to a classic inode or file set[1]. ZFS supports analogous behavior through the ZFS Intent Log (ZIL). Whether the ZIL would be utilized directly has not yet be determined but it does show that basic methodology of intent log has been proven.

At the time of intent log replay, the object or container associated with the HCE will be 'cloned' in preparation of extent pointer reassignment from the intent log. This is essentially an atomic copy of the object's extent pointers. The cloned object will be modified to account for epoch HCE+1 and the extents associated with HCE+1 will be combined with any relevant extents from the HCE such that proper reference counting is preserved. These activities are shown in Figure 2.

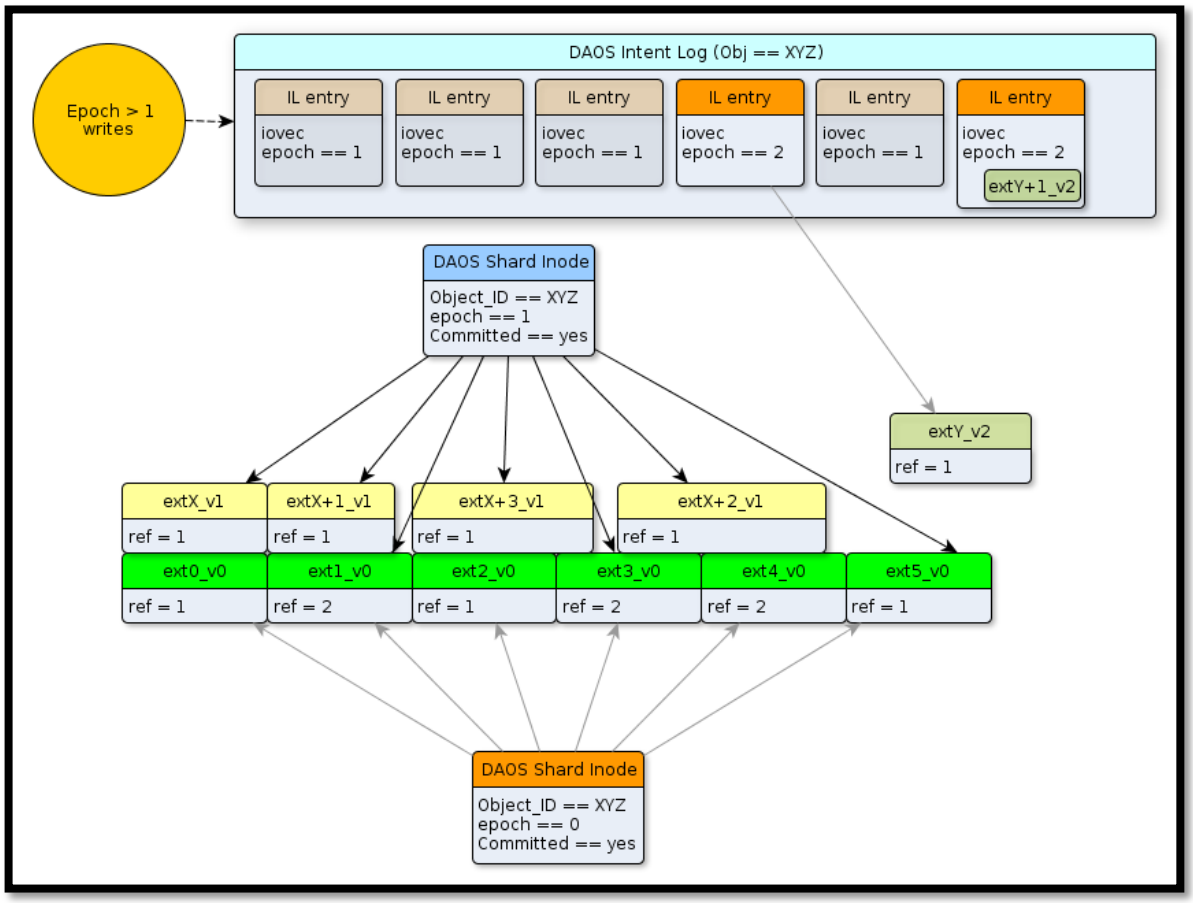

Figure 2 - Commit of Epoch 1

[1] File set refers specifically to ZFS which does not support explicit inode cloning. The ZFS CoW granularity exists at the file set. In VOSD, each DAOS container would exist in its own file set.

Epoch committal requires completion of intent log replay for the versions between the HCE and the epoch to be committed. Because the epoch commit procedure is an application driven task, it's possible that the commit epoch value greater than HCE + 1. In this case VOSD must ensure that all versions associated with a commit request have been replayed.

Figure 2 shows the manifestation of new object on behalf of object XYV, epoch 1 which accounts for the non-overwritten portions of epoch 1. In this state, the VOSD may still serve Object_XYZ@epoch_0 through the old object. In certain circumstances more than two versions may exist for a given object. This is certainly the case if snapshots have been taken on the object. At some point in the future, the unused versions of the object will be garbage collected. Garbage collection should be an efficient operation, requiring only metatdata updates in the form of version removal and unreferencing, or freeing, of data blocks. It is presumed that garbage collection will occur on non-snapshotted versions once all readers of the oldest version have departed.

A fully unified object XYZ is shown in Figure 3.

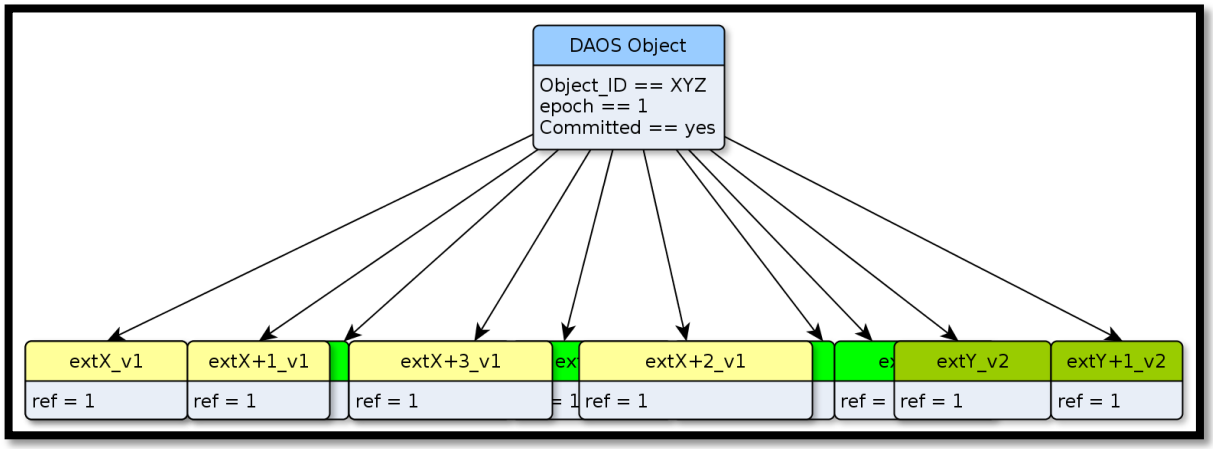

Figure 3 – Fully Unified Object

Unit/Integration Test Plan

1. Verify Residency of Single and Multiple Epochs for a given Object

The instantiation of an object with one or more epochs should be supported by the VOSD implementation. The unit test will show the existence of an object with a single version and prove that multiple versions may exist simultaneously. The test should show that a given object may be put and later retrieved. Further, it should show that several versions of the object may be put, each with some unique set of extents, and that those versions may be retrieved with their proper constituent extents.

2. Verification of Simultaneously Stored Epochs

This unit test should demonstrate the simultaneous storage of multiple epochs in a manner which maintains the affiliation of extents with their respective version.

3. Verification of Simultaneous Object Read and Write

The unit test demonstrates that writes of newer versions do not impact the correctness of reads from a previously committed version.

4. Verify Partial Overwrites Across Epochs

The 'partial overwrite' unit test proves that new version may be derived by applying one or more deltas to the last committed version. The test should demonstrate that no data copies were required in the construction of the new version.

5. Ensure Properly Functioning Object Cloning

A simple unit test should be capable of demonstrating object cloning within the VOSD in a manner similar to 'cp –reflink' functionality.

6. Verify that Epochs are Flattened Properly

The unit test proves that flattened objects expose and reference the proper extents and regions given some set of versions and corresponding deltas.

7. Verify the Correctness of Epoch Snapshots

Snapshotting an object should create an immutable version which is exempt from garbage collection. A unit test should take the following steps to prove that snapshots are maintained properly. First, a snapshot should be taken of some version X. Secondly, some number of versions should be

created for the object and extents should be applied to each version. Lastly, it should be shown that 2 explicit versions of the object remain after versions X+1 -> X_last have been merged – version X and version X_last.

Acceptance Criteria

1) Demonstrate success for all unit tests