High level design of a Lustre 2.0 Upgrade Tool

By Vitaly Fertman <vitaly_fertman@xyratex.com>
Date: 2011/05/01
Revision: 3.1

This document presents a High Level Design (HLD) of a Lustre 2.0 FS Upgrade Tool. The main purposes of this document are: (i) to be inspected by architects and peer designers to ascertain that high level design is aligned with Lustre architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of customers, architects, designers and developers.

0. Introduction

The on-disk format of Meta Data Server (MDS) file system has changed in 2.x Lustre FS. The existing 1.8 filesystems can be handled by the 2.x compatibility code, but not optimally from the performance point of view, they also have limitation on some features, like backup/ restore[1], changelogs[2].

The goal of the upgrade tool described in the document is to (i) to convert the MDS on-disk format of 1.8 Lustre FS to a native 2.x format; (ii) to restore the MDS 2.x Lustre FS metadata after backup/restore.

For the fid2path ioctl, we need to have Link EA for each inode, pointing to its parent, this EA is to be added to each inode.

To let users to use backup/restore, we need to ensure the Link EA parent pointer is valid after the restore. If parent does not have a real FID[3] assigned, its inode number is changed after the restore. Instead of fixing Link EA of all the files each time, it is better to assign a real FID to all inodes.

To let new allocated FID working properly, it needs to be added to the Object Index pointing to the inode number on this local filesystem. This Object Index(OI)[3] will always need to be updated after the backup/restore because the inode numbers are changed. At the same time, it is much cheaper than changing the LinkEA of all the files.

Lustre FS v2.x also has children FIDs stored in the directory entries. Whereas it is not mandatory, but rebuilding 1.8 directories in this way gives us a high performance gain on readdir and lookup: if FID does not exist in directory entry, we read inode and if its LMA EA is too large (over 128bytes in ldiskfs) and is stored in a separate inode, we read LMA EA. Thus, we could save 2 of 3 reads.

1. Definitions

OSD_root - this is the back end filesystem root.
MDD_root - this is the directory "ROOT" created in the OSD root, it is Lustre FS root.

Some previously used terms, just for the reference:
LinkEA - this is an inode extended attribute which keeps all the parent object FIDs.
LMA EA - this is an inode extended attribute which keeps this object FID among other attributes.

2. Requirements

Convert on-disk data structures on MDS of Lustre FS 1.8 to Lustre FS 2.x ones in-place.

After upgrading, files will have:

- true FIDs, stored in LMA EA;
- parent FIDs, stored in Link EA;
- children FIDs, stored in directory entries.

After upgrading Object Index will have:

- FIDs pointing to the proper inode numbers;

3. Design highlights

The migration tool will be done in-kernel, triggered at mount time if a special mount option is given. The mount completes only after the upgrade is done, therefore the system is left unavailable in the meanwhile. If user decides to interrupt the mount, it stops the progress in the middle, i.e. only a part of files are upgrated, and fails the mount. However, the code could be later extended to an on-line conversion tool, on-line fsck, etc.

Several MDD threads are traversing the semantic tree. Each thread takes a directory entry, fix the child and parent-child links, Object Index entry.

These threads populate a TODO list with directories being handled to let other threads handle other directory entries in the same directory in parallel.

4. Functional specification

A new mount option “upgrade”.

Once given on MDS node, the on-disk format rebuild is triggered. It includes:

- assigning true FIDs to all inodes;
- storing new FIDs in LMA EA;
- storing parent FID in Link EA;
- adding children FIDs to directory entries;
- adding Object Index records for new FIDs.

A new mount option “restore”.

Once given on MDS node, the on-disk format rebuild is triggered. It includes only:

- adding Object Index records for new FIDs.

This mount option to be used after the backup/restore feature.

A Sequence client on MDS.

To assign real FIDs to files, these FIDs need to be properly allocated, through a Sequence client. It usually runs on client nodes, but we need to run it on MDS node for our needs.

The new rebuild threads.

The threads traverse the semantic tree, each thread rebuilds a directory entry and the object referenced by this entry as well as the back reference on the parent from the object. The threads populate the TODO list with information about the object being rebuilt and the position in the object, i.e. the entry hash. It lets other threads to proceed with these objects in parallel.

The object rebuild.

Once an object is loaded to memory, it must already have the proper FID assigned which should not be changed anymore. Preferably, even readdir should already return the proper FID in read directory entries to avoid an ambiguity. Therefore, the following is to be done right on readdir or lookup:

- to assign a new real FID to the child object;
- to store this FID in LMA EA of the child object;
- to add this FID to a directory entry of the current parent object;
- to add Object Index record for new FID.

The back reference to the parent in the LinkEA could be recovered separately.

The above needs some changes in the dt_it_ops iterator interface and in the dt_index_operations index interface to let each layer to rebuild each own part of the objects.

The dt_it_ops iterator interface, a new method ->rebuild().

This method is similar to the ->rec() method, i.e. it does the directory entry read, but also requests the underlying layer to rebuild the directory entry and the child object referenced by it, as described above, so that the read entry already had a proper rebuilt FID.

The dt_index_operations interface, a new method ->dio_rebuild().

This method is similar to ->lookup(), but also requests the OSD layer to rebuild the directory entry and the child object referenced by it, as described above, so that the read entry already had a proper rebuilt FID. In fact, this method does the same job as the ->rebuild() method above, but the object is referenced by a name in parent, in contrast to the hash in the ->rebuild().

5. Logical specification

The MDD layer, the TODO list.

This list contains the job to be done, each TODO item needs to keep the following info:

todo_item_parent : a pointer to the parent TODO item;
todo_item_object : a pointer to the file object described by this TODO item;
todo_item_hash : the position in the file object, the hash of the directory entry;
todo_item_ref_count : the counter of the references taken on the TODO item;
todo_item_list : the link to the TODO list;

Only directory objects are added to the TODO list, so the position is defined as the directory entry hash. The item also has a parent reference and a reference counter of existing TODO children. The reference counter is also taken by the thread handling this item.

The MDD layer, mdd_prepare(), the upgrade tool start point.

This is the place where the job starts, mdd_prepare(). It does the following:

- opens the OSD_root object;
- looks up the name of MDD_root ("ROOT") in the OSD_root object through a special rebuild method dt_index_operations->dio_rebuild();
- starts a Sequence client;
- creates a TODO item for the MDD_root object, adding it to the TODO list;
- start the threads and waits for their completion.

The MDD_root object has a fixed FID in 2.x fs, known on MDD layer only:

[MDD_ROOT_INDEX_OID, FID_SEQ_LOCAL_FILE]

notice, that this is a local FID, not a normal real FID. This FID is passed to the underlying OSD layer in ->dio_rebuild() so that the object is to be rebuild with it, if no real FID is assigned yet.

The MDD layer, the rebuild threads.

The threads traverse the semantic tree in child-first order. Each thread:

- takes a TODO item, removing it temporary from the TODO list;
- takes the next, not handled yet, directory entry in an object by means of the hash stored in the TODO item (todo_item_hash);
- reads the directory entry through a special rebuild method: dt_it_ops->rebuild();
- adds the curent directory object FID in the Link EA of the child object, if there is no such yet;
- checks if this entry points to a directory, if so the thread switches to it, returning the parent directory to the TODO list to let other idle threads to handle the next entry; the hash in the TODO item is to be updated to the last handled entry by this time.

TODO list is always populated from the head to the most recent items, child objects, to be handled first.

Once a directory is rebuilt completely with all its children, the thread takes its parent following the parent link in the TODO item (todo_item_parent). If the parent is not linked to the TODO list, (via todo_item_list) it means another thread is handling it or its recovery is completed (the TODO item still exists just because its child was in progress, it is controlled by todo_item_ref_count), the parent is skipped, and the thread takes its parent, and so on until a TODO item in the TODO list is found or the MDD_root is reached. If no TODO item is taken for the processing so far, the thread waits until the TODO list gets anything to do.

A Sequence client starts at MDD layer and is shared between all the threads, each thread allocates a new FID in advance, passes it to the iterator method ->rebuild(). If the handled object has the same FID as the last allocated one, the FID has been used and another one is to

be allocated for the next ->rebuild(). At the end, there is a leak of allocated FIDs equals to the amount of rebuild threads – not a problem.

XXX: all the directories are handled in parallel until only 1 is left, which is handled by 1 thread only. To be improved later.

The OSD layer, the iterator method ->rebuild() in the EA iterator.

The OSD layer is responsible to return the object with its FID to upper layers wherever the FID is stored. This is the proper place to do the object rebuild. This method is an analog of the ->rec() method, i.e. it does the directory entry read, but also rebuilds LMA, OI, directory entry with a new FID given by the caller:

1. calls osd_ea_fid_get();
   - osd_ea_fid_get() creates LMA with the new FID, if no FID is assigned to the child object yet;
2. takes osd_inode_id and FID found by osd_ea_fid_get() and passes them to osd_oi_rebuild() and to osd_direntry_rebuild();
   - osd_oi_rebuild() checks if the OI for the given FID exists and points to the same osd_inode_id, replace/add OI entry otherwise;
   - osd_direntry_rebuild() checks if direntry has no FID or has IGIF, replaces the entry with a new one which has the given FID added.

A special handling of the ".." entry is needed if the directory being handled is a local file, i.e. the MDD_root directory. The entry points to the OSD_root which does not need to have LMA EA – avoid LMA EA rebuild in osd_ea_fid_get() for this case.

The OSD layer, the index operations method ->dio_rebuild() in the EA index.

This method is an analog of the ->lookup() method, but also rebuilds LMA, OI with the FID given by the caller. It works the same as the iterator method ->rebuild() described above, but without the directory entry rebuild.

This method is used for the MDD_root object rebuild only, so the directory rebuild is skipped for now. It could be easily extended to let any object rebuild this way – it is enough to check if the current object is identified by a local FID and skip the directory rebuild if so.

Note, LMA EA is also added to the MDD_root object. It is done to simplify the rebuild of the ".." entries of the MDD_root children, and of the "." entry of the RMDD_root object. Without it, we can find the MDD_root object, but OSD layer is not aware about the fixed MDD_root fid and therefore unable to build the proper fid.

6. State

No new object states appear, once the object is loaded it cannot be changed. Thus, all the rebuild job is done on the directory entry read and the lookup code paths, so that in-core object state would already appear in the converted 2.x format.

The new TODO item has the following states: pending, in-progress, completed.

In-progress state: it means a thread is handling this item, the item is NOT in the TODO list, but it has a taken reference (todo_item_ref_count) which is an indicator the recovery is being done.

If the handling thread switches to its child, the item is moved to the pending state: added to the TODO list. An extra reference is taken indicating a child item exists and points to this one.

If the handling thread has completed the rebuild of this item and no more rebuild is needed, the item is moved to the completed state: reference is put (todo_item_ref_count).

Pending state: it means this items still needs some rebuild to be done, but no thread is handling it currently. The item is in the TODO list and a reference is taken (todo_item_ref_count), what means some recovery is still needed.

If a thread takes the item for handling, it is moved to the In-progress state: removed from the TODO list.

Completed state: it means the rebuild of the item has completed, the item is not in the TODO list and has no extra reference indicating a rebuild is needed (although the item may have a children references).

6.1. Lifetime.

A new TODO item is allocated, when the processing thread detects a child is a directory and it is switching to it. The item gets to the in-progress state immediately, i.e. not get added to the TODO list but gets a taken reference at this time.

A TODO item is deallocated, when it is in the completed state and there are no more taken references, i.e. no more children pointing to this item. A reference to the parent is put at this time, which may lead to a parent deallocation at this time.

6.2. Concurrency control

It is possible that several thread will try to access a TODO item at the same time, or will try to modify the same object, thus some concurrency control is needed.

To maintain the TODO list consistency, a spinlock is needed. A new one is to be added. This spinlock also controls the transitions of the TODO item from one state to another.

Objects added to the TODO list just need to be pinned. It is enough to check if the object is still alive at the handling time. However, it will be alive as this is an off-line tool so far.

Thread which takes a TODO item for handling, removes it from the TODO list, so no other thread can access it in meanwhile (only directories can be added to the TODO list). This will probably change in time to let large directories to be handled by several threads at a time.

Files could be reached from different directories so a race may appear. The serialization is

done through osd_object.oo_sem semaphore.

Directory entry change is also protected by parent's osd_object.oo_sem semaphore.

Another similar race may appear with Link EA rebuild, it is done on MDD layer and the serialization is done through dt_object_operations locking operations which result in taking the same osd_object.oo_sem semaphore on the OSD layer.

Only one thread is handling 1 FID at a time, so there is no need in extra serialisation per-FID, the OI file access is serialized by the underlying fs.

7. Use cases

7.1. Scenarios

Scenario	The full FS upgrade or restore.
Business goals	The upgrade to 2.x format is needed to get a better performance out of 2.x fs and get an opportunity to use changelogs and backup/restore.
Relevant quality attributes	Availability, performance, interoperability, scalability.
Stimulus	The mount option “upgrade” or “restore” is given.
Stimulus source	Administrator request.
Environment	MDS
Artifact	MDD rebuild threads are working temporary, once the job is done before the mount completes.
Response	The system will be off-line and therefore unavailable during some period of time until after MDD rebuild threads complete. Once complete, the FS gets mounted; its readdir and lookup performance can increase up to 3 and up to 2 times accordingly; ChangeLog and restore/backup features become available.
Response Measure	Most customers do not want a system to be off-line for more than 12hours. It would be great to be within this limitation.
Questions and issues	

7.2. Failures

An object is skipped as well as all its children if some problem appears during its rebuild. To inform users that some problems appeared during the upgrade, we can also fail the mount at the end.

To not leave the system in a broken state after a failure, the object FID, LMA, OI rebuild is to

be done atomically, i.e. within the same transaction. Directory entry rebuild and LinkEA could be done separately.

8. Analysis

8.1. Scalability

To get the best performance the amount of threads could be equal to the amount of CPUs.

Some more performance speedup could be achieved by analyzing the amount of the disks used in the MDS FS and trying to launch 1 thread per-disk at least. Any thoughts how to do it during the semantic traverse?

8.2. Rationale

An alternative implementation could be a user space tool, probably an integration to the fsck. However, it has several drawbacks:

- there is no good expertise in the fsck code;
- a separate utility requires too much code to be ported from the kernel;
- the speed of the conversion is higher if being implemented right in the kernel;
- the in-kernel tool lets us to extend it to an on-line tool with time.

9. Deployment

9.1. Compatibility

9.1.1. Network

After upgrade, the objects are addressed by changed FIDs, therefore all the replayable clients are to be evicted after the upgrade to drop their already invalid cache.

9.1.2. Persistent storage

No issues. The on-disk object is either of 1.8 or 2.x format, after the convertion it is only of 2.x format. Lustre FS v2.x started on 1.8 FS already handles both.

9.1.3. Core

No issues. In-core object can appear only in 2.x format after/during the upgrade.

The iterator interface dt_it_ops gets a new method: ->rebuild(). It rebuilds a directory entry being iterated and the object pointed by this entry.

The index operations dt_index_operations get a new method: ->dio_rebuild(). It rebuilds an object found through lookup in the given directory.

10. References.

[1] Backup/Restore
http://wiki.lustre.org/manual/LustreManual20_HTML/
BackupAndRestore.html#50438207_pgfId-1292693

[2] ChangeLogs
http://wiki.lustre.org/index.php/Architecture_-_Changelogs
http://wiki.lustre.org/images/7/72/Changelog-hld.pdf

[3] FID
http://wiki.lustre.org/images/1/10/Cmd-fid-hld.pdf
http://wiki.lustre.org/images/4/4e/Cmd-fld-hld.pdf

[4] Object Index
http://wiki.lustre.org/images/a/a0/Iam-hld.pdf