## SCOPE OF WORK

### Project 1. Layout Enhancement Design

#### *Technical Description and Approach*

Lustre file layouts represent how file data is distributed over OSTs. Only simple RAID0 (striping) layouts are supported today, and enhancements are required to implement future features such as Data on MDS, Data Replication, live Data Migration, and RAID1/5/6 or erasure coding. The Layout Enhancement (LE) project is therefore a prerequisite for the Data on MDS and Replication projects proposed here. LE was separated from these subsequent projects in order to ensure the design not only satisfies requirements for these two efforts, but also anticipates future developments. Design goals are listed below.

- Extend the Lustre layout to include the following formats:
    - N-way stripe replication. This is a requirement for the Data Replication project. It generalizes RAID0+1 to allow multiple replicas of the data in a Lustre file. This will ensure that data can remain available in the face of multiple OST failures - e.g. multiple short- or long-term OST failures.
    - RAID5/6 and erasure codes. These anticipate future developments to enable more space-efficient replication techniques.
    - Layout extents. This allows different layouts in different extents of a Lustre file. It can be used to enable wider striping as files grow in size, to prevent inconsistent ENOSPACE errors as individual OSTs become full, and to enable incremental migration, replication, and HSM restore.
- Determine additional requirements for algorithmic layouts – e.g. CRUSH.
- Extend the layout locking protocol to ensure layout changes can be effected while a file is actively being accessed. Also, determine client-side changes required to allow layout changes to be requested within the I/O stack. These are required for the Data on MDS project to migrate small files off the MDS when they grow larger, and needed by the Replication project to enable layout changes in response to OST availability. They will enable future features such as incremental HSM restore and live OST rebalancing.
- Determine a new protocol strategy for handling large layout representations. The current strategy can result in grossly inefficient network buffer utilization for extremely large layouts and assumes layouts are always transferred in their entirety.
- Compatibility and version interoperation. The design work must address requirements to support rolling upgrades.

## Project 2. CLIO Simplification Design

### *Technical Description and Approach*

The Lustre client implementation for the IO path (called CLIO) is responsible for issuing RPC commands for reading and writing data to the OSTs. CLIO was reconstructed in Lustre 2.0 for cross-platform portability. The implementation is too complex for the current usage, thus making the code hard to understand and maintain. Moreover, from the performance benchmarks, the IO performance of 2.x clients is significantly lower than 1.8 clients.

These major efforts will be made for this project:

1. Simplify the implementation of cl_lock by implementing a cache-less lock. This will affect the whole IO stack regarding lock requesting, lock management, and page cache. After this is done, cl_lock should become much easier to understand and maintain.
2. Deprecate liblustre support which has been unused for several years and was only kept in Lustre as an example of a non-Linux IO architecture.
3. Remove the stub Mac OSX and Windows client support. Once complete, the CCC layer would disappear and the Lustre client should be more compatible with the Linux kernel VM and VFS APIs.
4. Further simplify the IO stack by removing the vvp and lovsub layers. These layers were added for platform portability. However, since functional ports to new OSs is unlikely at this point, we propose to remove these layers to simplify the overall IO stack. This will simplify the code and increase the likelihood of Lustre client inclusion into the Linux kernel.
5. Realize performance improvements resulting from code simplification and abstraction removal.
6. Optimize the use of file ioctl() calls. For example, have LL_IOC_DATA_VERSION use CLIO interfaces and remove access to log_stripe_md at the llite layer.
7. Remove obsolete OBD callbacks such as obd_brw(), obd_punch(), etc.
8. Explore whether VFS reads and writes can be optimized at the expense of mmap performance.

## Project 3. Removal of Dead Code in Lustre (Implementation)

### *Technical Description and Approach*

Ongoing restructuring of the Lustre source code has created large swaths of unreachable code and unused data. At the same time, the layering and complexity of Lustre makes this dead code and data difficult to identify during restructuring, adding unnecessary complexity to ongoing development and maintenance. We propose a concerted effort to address this issue.

This project will identify and remove code from the Lustre codebase which is no longer being used. This will be achieved by using static code analysis tools to isolate areas of dead code. Engineers will then hold inspections to understand why the code is dead and on how to best remove the code with minimal impact to other on-going projects.

Some areas that will be targeted:

- OBD methods and handlers
- /proc files and structures
- libcfs module APIs
- Utilities
- liblustre

The initial rough estimation is that over 15,000 lines of code could be removed from the Lustre codebase as a result of this effort.

## Project 4. Code Documentation (Implementation)

### *Technical Description and Approach*

Many subsystems of the Lustre code do not have sufficient internal documentation (code comments) to describe the implementation sufficiently, and some of the comments are out of date. In order to facilitate developer understanding of the code; and in turn improve their ability to enhance and fix code in a robust manner, a project will be undertaken to add and improve Doxygen-formatted comments within select subsystems. Code commenting best practices will be added/updated to the Lustre Coding Guidelines wiki page.

Comments added to each file will include an introductory comment block describing the high-level functionality (at least two or three paragraphs) in the updated subsystem. A comment block describing each function will be added to all functions more than a few lines in length, including input/output arguments, locking requirements, code caveats and complexities. Additional comments will be added inline with the code as necessary.

The Lustre subsystems to be commented in Phase 1 of this project will be lustre/lod, lustre/osp, and lustre/ofd. These subsystems were chosen because while they have had significant rework recently, there were large parts of the code copied from old Lustre code (lov, osc, and obdfilter respectively) that were lacking in clear documentation. This project will prove beneficial from the review of the existing comments, as well as benefit others that are not familiar with the significant changes that this code has undergone.

## Project 5. Data on MDS Design

### *Technical Description and Approach*

Lustre performance is currently optimized for large files. This results in additional RPC round-trips to the OSTs, which hurt small file performance. This project aims to correct this deficiency by allowing the data for small files to be placed on the MDS so that these additional RPCs can be eliminated and performance correspondingly improved.  Used in conjunction with DNE, this will preserve efficiency without sacrificing horizontal scale.

System administrators will set a layout policy that determines a minimum file size below which files will be contained entirely on the MDS.  Files that grow beyond this size will be migrated onto OSTs. The layout policy can be generalized to include further size breakpoints with different default layouts. This will allow progressively wider striping as files continue to grow, using the same underlying restriping mechanisms developed to migrate small files off the MDS.  Design goals are listed below.

- Unified target.  Unified request handling for OSTs and MDTs is required to allow data operations through the MDT to the underlying OSD.
- Client I/O.  Inclusion of the MDC as part of the client-side I/O stack.
- Dynamic data migration.  Moving file data from the MDT onto OSTs while I/O is active.
- Dynamic stripe allocation.  Allocating new OST objects on demand while I/O is active.
- Administration.  Generalization of existing default layout policy to specify restriping at different file sizes.

## Project 6.  Replication Design

### *Technical Description and Approach*

Lustre availability and resilience relies entirely on the availability and resilience of its backend storage devices.  Replication mitigates this dependency by specifying a mirroring file layout for data across multiple OSTs so that data remains available in the event of device failure.  Data integrity can be checked and repaired by comparing mirrors.

The design will anticipate a phased implementation.  In the first phase, replica OST object generation will be performed on otherwise idle files by a generic userspace replication utility.  This will be used both to turn a non-replicated file into a replicated file and to restore the required level of replication to files stored on OSTs that have failed or otherwise become unavailable.  In this phase, replicas of overwritten OST objects will be discarded.  Immutable files will therefore retain resilience after initial replica generation.  Append-only files will retain resilience until appended extents are updated by the replication utility. However, replication for overwritten files will have to be regenerated completely.  In the second phase, replica OST objects will be maintained synchronously. This will allow replication immediately on file creation and replication to be preserved on overwrites.

Replica OST object allocation algorithms will be developed to guarantee fault isolation between replicas and load balance replica generation over all available OSTs.  This will ensure single hardware and software failures cannot affect multiple replicas and that repair speed after such a failure scales with aggregate OST bandwidth.  Replication will be selectable on a per-file basis, and described by new layout formats designed in the LE project. Existing default layout policy will be able to use these to control the level of replication. Design goals are listed below.

- Parallel replication utility. This will set or restore the requested level of replication on a set of Lustre files by allocating and initializing new replica OST objects and performing a layout switch to include them in the original file layout when required, or bringing replicas in appended extents up to date on append-only files. The phase 1 design will perform this on idle files and abandon or restart replication if the file is overwritten. The phase 2 design will perform replication repairs on files that are actively being written.
- Replica de-clustering and fault isolation. Configuration metadata will be added to assign OSTs to fault domains so that replica allocation can avoid single points of failure. Replica allocation will also ensure replicas are dispersed over all available OSTs to ensure replica regeneration on OST failure can proceed at full system bandwidth.
- Client I/O stack. The client I/O stack will be altered in the phase 1 design to discard replicas on overwrite, maintain the replication extent within append-only files and to read from replicas when primary OSTs are disabled, RPCs to the primary OST return failure; or to verify data integrity on request. In phase 2 the client I/O stack will be altered to clone writes across all replicas and to handle write failures appropriately.
- Administration. Existing default stripe administration methods will be extended to handle replication. A utility that implements a simple delayed replication policy triggered by the file system ChangeLog will be included to perform delayed replication for the phase 1 design. An OST repair utility will scan the file system namespace for files that included OST objects on a failed OST and invoke the parallel replication utility to repair them.