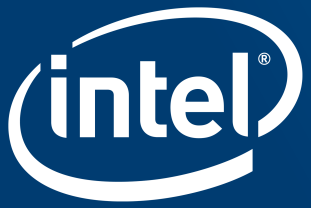


Features In Development for Lustre* 2.10 and Beyond

Oleg Drokin

High Performance Data Division, Intel

JLUG2016, Tokyo

Ongoing Performance and Functional Improvements

2.10 has the potential to be a very interesting release

- ZFS feature and performance improvements
- Multi-Rail LNet for improved performance and reliability
- Composite File Layout for improved performance and ease of use
- Data-on-MDT for improved small file performance/latency
- File Level Redundancy (read-only) for improved reliability and performance
- Miscellaneous smaller features

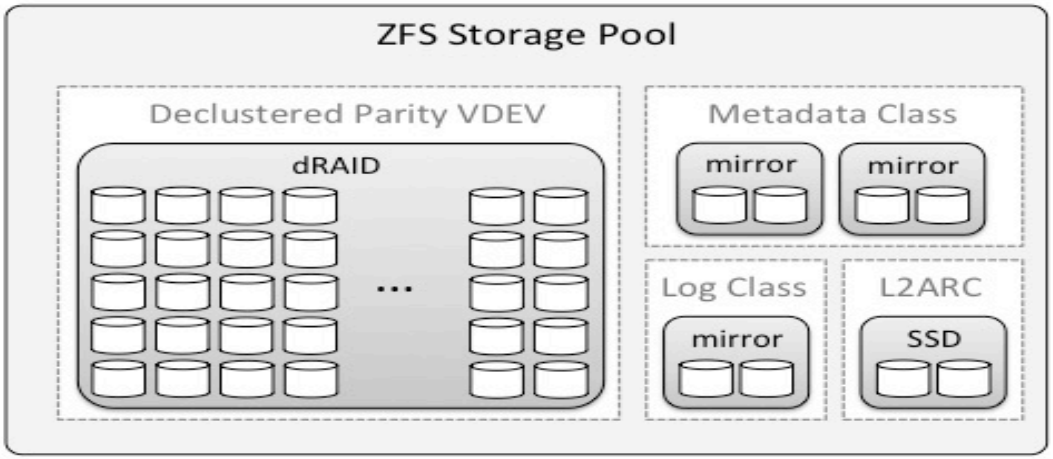
Ongoing research projects

Potential future development projects

ZFS Enhancements (Intel, LLNL 2.10+)

Changes for using ZFS more efficiently

- Improved file create performance (Intel)
- Snapshots of whole filesystem (Intel)



Changes to core ZFS code

- Inode quota accounting (Intel)
- Multi-mount protection for safety (LLNL)
- System and fault monitoring improvements (Intel, HPE)
- Large dnodes for improved extended attribute performance (LLNL)
- Reduce CPU usage with hardware-assisted checksums, compression (Intel)
- Declustered parity & distributed hot spares to improve resilvering (Intel)
- Metadata allocation class to store all metadata on SSD/NVRAM (Intel)

Multi-Rail LNet (Intel, SGI* 2.10)

Allow LNet across multiple network interfaces

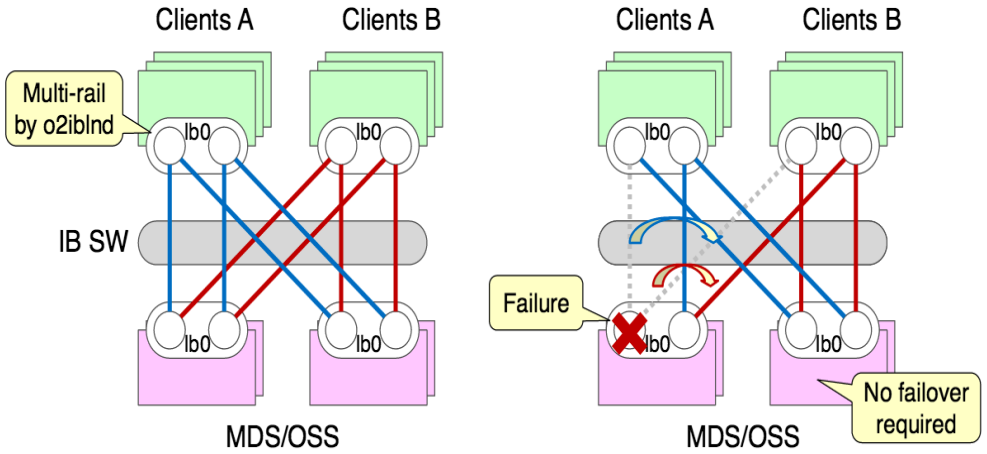
- Supports all LNet networks – LNet layer instead of LND layer
- Allows concurrent use of different LNDs (e.g. both TCP and IB at one time)

Scales performance significantly

- Scaling limits currently being tested

Improves reliability

- Active-active network links between peers



Composite File Layouts (Intel, ORNL 2.10+)

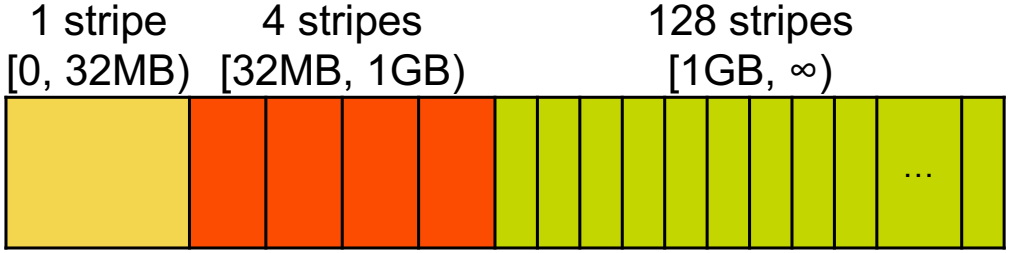
Composite File Layout allows different layout based on file offset

- Provides flexible layout infrastructure for upcoming features
  - File Level Redundancy (FLR), Data-on-MDT (DoM), HSM partial restore, etc
- Layout components can be disjoint (e.g. PFL) or overlapping (e.g. FLR)

Progressive File Layout (PFL) simplifies usage for users and admins

- Optimize performance for diverse users/applications
- One PFL layout could be used for all files
- Low stat overhead for small files
- High IO bandwidth for large files

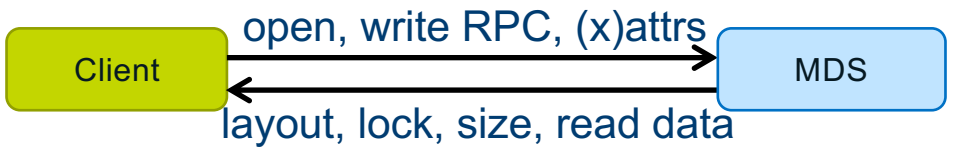
Example progressive file layout with 3 components



Improved Small File Performance (Intel 2.11)

Data-on-MDT optimizes small file IO

- Avoid OST RPC overhead (data and lock RPCs)
- Use high-IOPS MDT storage (mirrored SSD vs. RAID-6 HDD)
- Avoid contention with streaming IO to OSTs under load
- Prefetch file data with metadata
- Size on MDT for regular files
- Manage MDT space usage by quota



Small file IO directly to MDS

Complementary with DNE 2 striped directories and PFL layouts

- Scale small file IOPS with multiple MDTs
- Use PFL to extend larger files from MDT to OST storage

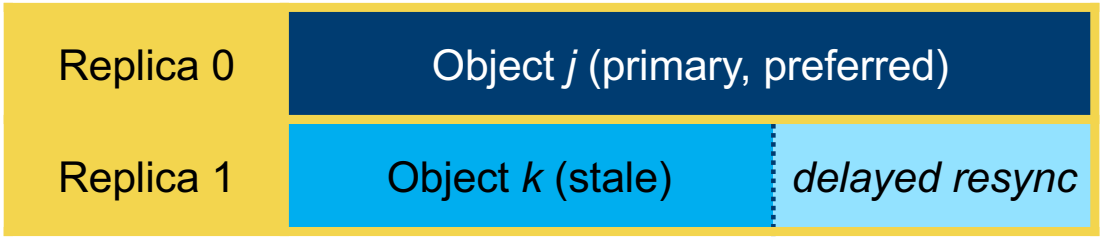
File Level Redundancy (Intel 2.10+)

Provides significant value and functionality for both HPC and Enterprise use

- Select layout on a per-file/dir basis (e.g. mirror all input data, one daily checkpoint)
- Higher availability for server/network failure - finally better than HA failover
- Robustness against data loss/corruption - mirror or M+N erasure coding for stripes
- Increased read speed for widely shared files - mirror input data across many OSTs

Replicate/migrate files between storage classes

- NVRAM->SSD->HDD
- Local vs. remote replicas



FLR Phased Implementation Approach

Can implement Phases 2/3/4 in any order

Phase 0: Composite Layouts from PFL project (Intel, ORNL 2.10)

- Plus OST pool inheritance, Project/Pool Quotas

Phase 1: Delayed read-only mirroring – depends on Phase 0 (Intel 2.11)

- Manually replicate and migrate data across multiple tiers

Phase 2: Integration with policy engine/copytool - with/after Phase 1

- Automated migration between tiers based on admin policy/space

Phase 3: Immediate write replication – depends on Phase 1 (Intel 2.11?)

Phase 4: Erasure coding for striped files - with/after Phase 1 (2.12?)

- Avoid 2x or 3x overhead of mirroring files

FLR with Erasure Coding (2.12?)

Erasure coding provides redundancy without 2x or 3x overhead of mirrors

Add redundancy component to existing striped files *after* write is finished

- Can add parity component to any existing RAID-0 file

Suitable for striped files - add N parity per M data stripes (e.g. 16d+3p)

- Multiple parity groups avoids IO bottlenecks, CPU overhead of too many parities
- Should take failure domains into account (avoid data and parity on same OSS)
  - e.g. split 128-stripe file into 8x (16 data + 3 parity) with 24 parity stripes

dat0	dat1	...	dat15	par0	par1	par2	dat16	dat17	...	dat31	par3	par4	par5	...
0MB	1MB	...	15M	p0.0	q0.0	r0.0	16M	17M	...	31M	p1.0	q1.0	r1.0	...
128	129	...	143	p0.1	q0.1	r0.1	144	145	...	159	p1.1	q1.1	r1.1	...
256	257	...	271	p0.2	q0.2	r0.2	272	273	...	287	p1.2	q1.2	r1.2	...

Miscellaneous Features

Server-side IO advice - `ladvise` (DDN* 2.9+)

- Tunables per file+extent to manage OSS cache (`willread`, `dontneed`)
- Set ZFS blocksize per file (maybe also compression level/type, checksum, ... ?)

Client write DLM lockahead (Cray* 2.10)

- Allow libraries/apps to prefetch write locks for striped or arbitrary IO patterns

Client code cleanups (ORNL, Intel, Cray 2.10+)

- Update to match upstream kernel, port patches to/from kernel
- RHEL weak symbol versioning, patchless server kernels

Project Quotas (DDN 2.10)

- Allow quota tracking on directory subtrees independent of UID/GID

Advanced Lustre* Research
Intel Parallel Computing Centers

Uni Hamburg + German Client Research Centre (DKRZ)

- Client-side data compression
- Adaptive optimized ZFS data compression

GSI Helmholtz Centre for Heavy Ion Research

- TSM* HSM copytool

University of California Santa Cruz

- Automated client-side load balancing

Johannes Gutenberg University Mainz

- Global adaptive IO scheduler

Lawrence Berkeley National Laboratory

- Spark and Hadoop on Lustre

Potential development proposals for the future...

High performance ZFS on all-flash storage

- Reduce memory access from compression and parity with NUMA affinity

Tiered storage with Composite Layouts and File Level Redundancy

- Integration with RobinHood to manage migration between tiers, rebuild replicas

Metadata Redundancy via DNE2 distributed transactions

Local persistent cache on client with fscache or local OSD

- Use FLR to ensure availability in case of client failure

Optimized IO fast path for client-on-OSS for copytool or local OSD

Legal Information

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at http://www.intel.com/content/www/us/en/software/intel-solutions-for-lustre-software.html.

Intel technologies may require enabled hardware, specific software, or services activation. Check with your system manufacturer or retailer.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

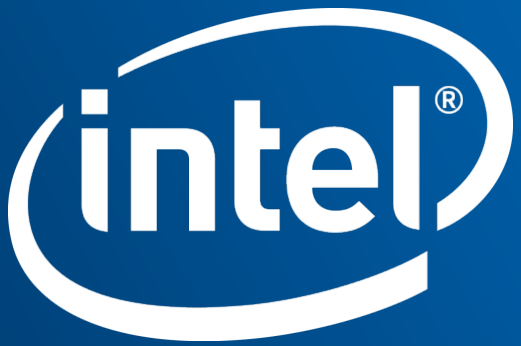
The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Intel, the Intel logo and Intel® Omni-Path are trademarks of Intel Corporation in the U.S. and/or other countries.

* Other names and brands may be claimed as the property of others.


intel

Backup Slides

File Level Redundancy details

FLR Phase 1: Replica File Layout Options

Redundancy based on overlapping composite layouts

- Layout extents with overlapping `{ lcme_extent_start, lcme_extent_end }`
  - Each component a *plain* layout (currently RAID-0, but DoM possible in the future)
- Most obvious usage is mirror of single-striped files
- Can have multiple replicas, as many as will fit into a layout xattr
  - 500 single-stripe components about same size as one 2000-stripe RAID-0 layout
- Can replicate RAID-0 files, stripe count can be different, stripe size must match
  - For example, if SSD or local OST count doesn't match HDD or remote OST count
- Can also replicate PFL files by having multiple overlapping components

FLR Phase 1: Creating Replicas/Mirrors

Replica initially created by userspace process

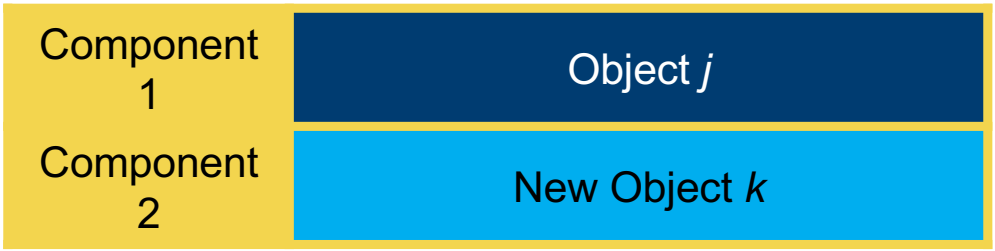
- Replica created or sync'd some time after file finishes being written

Any kind of file copy mechanism is OK to use for the replica

- Can be driven directly by user similar to `lfs migrate` or via HSM copytool
- Can use policy engine (e.g. RobinHood) to tune by path, user, size, age, etc.

Replica file copy attached as composite layout with overlapping extent(s)

- Move copy layout as replica component
- File now robust against OST failure/loss
- Can make replica in different storage tier



FLR Phase 1: Read Delayed Replication/Mirrors

Client has no idea how replica was created

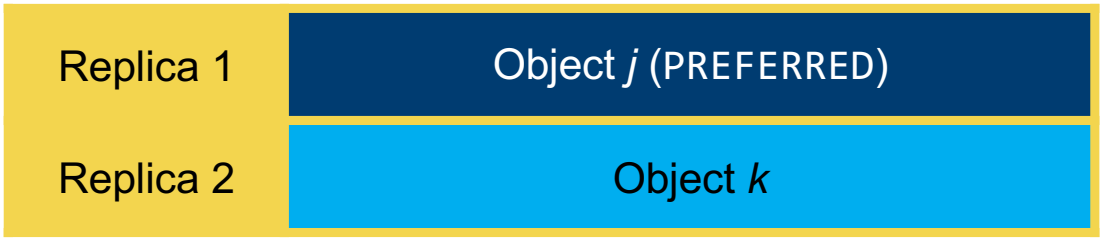
- Only needs to be able to read the components at this stage

File can be read by any composite-file-aware client

- Normal file lookup fetches composite layout that describes all replicas/objects
- Read lock any replica OST objects to access data

If Read RPC times out, retry with some other replica of that extent

- Read policy can be tuned



FLR Phase 1: Selecting Read Component

Client selects component objects to read based on available extent(s)

- Select component extent(s) that match current read offset, resolve to OST(s)
- Prefer component(s) marked `PREFERRED` by user/policy (e.g. SSD before HDD)
- Skip any OST object(s) which are marked inactive
- Prefer OST(s) by LNet network if OSTs local vs. remote
- If few OSTs left or large file - read same data from each OST to re-use cache
  - Pick components by offset (e.g. component = (offset / 1GB) % num_components)
- If many OSTs left or small file - read data from many OSTs to boost bandwidth
  - Pick components by client NID (e.g. component = (client NID % num_components))

FLR Phase 1: Writing to Read-only Replicas

Write synchronously marks all but one `PRIMARY` replica `STALE` in layout

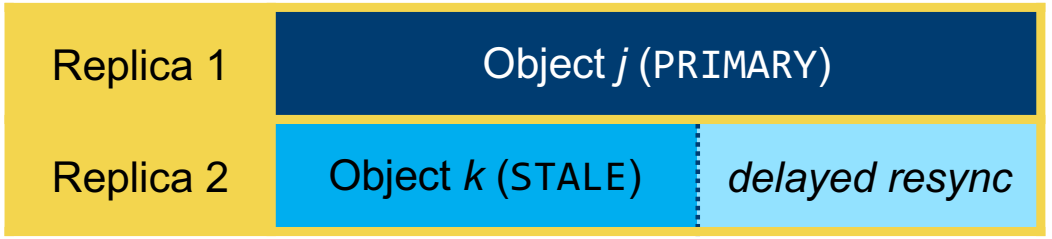
- This is not worse than today when there was never any replica, `STALE` replica is a backup
- Write lock all replicas - MDT `LAYOUT` and OST `GROUP EXTENT` on all objects to flush cache
- Set `PRIMARY` flag on one replica, `STALE` flags on other(s), add `STALE` record to ChangeLog

All writes are done only on the `PRIMARY` component(s) at this point

Resync is done after write finished in the same way initial replica was created

- Can incrementally resync `STALE` replica
- Clear `STALE` layout flag(s) when done

Can read replicas again as normal



FLR Phase 2: Integration with HSM File Layout

Merge HSM xattr into normal layout as a new file layout type

- Store HSM archive identifier as Lustre xattr instead of FID stored in HSM
- Can have multiple archive copies of a single file (e.g. local, offsite, versions?)

Restoring part of very large file doesn't need to block client(s) until restore done

- Truncate unwritten end of current component, add a new component after it
- Continue restore in second component (maybe wider striped?) like PFL
- Client can start using first component instead of waiting for whole file restore
- Can safely abort restore of last component without invalidating all restored data

FLR Phase 2: Integration with Policy Engine

Leverage HSM Policy Engine, copytools to replicate/migrate across tiers

- Functionality starting to appear in RobinHood v3
- Replicate/migrate by policy over tiers (path, extension, user, age, size, etc.)
- Release replica from fast storage tier(s) when space is needed/by age/by policy
- Run copytools directly on OSS nodes for fastest IO path
- Partial restore to allow data access before restore or migration completes

Migrate data directly by command-line, API, or job scheduler if needed

- Pre-stage input files, de-stage output files immediately at job completion

All storage classes in one namespace means data always directly usable

FLR Phase 3: Immediate Write Replication

Client generates write RPCs to 2+ OSTs for each stripe of the file

- Data page is multi-referenced: does not double memory but *does* double IO
- Most files will not have any problems, no need for resync in most cases

OST write failure requires sync RPC to MDT to mark component `STALE`

- `MDS` writes a ChangeLog record for `STALE` component
- No more writes to that component until it is repaired and no longer `STALE`

Client failure during write has MDS mark non-`PRIMARY` components stale

- `STALE` components resynced from userspace policy engine using ChangeLog

FLR Phase 4: Erasure Coded Files

Erasure coding provides redundancy without 2x or 3x overhead of mirrors

Add redundancy component to existing striped files *after* write is finished

- Can add parity-only component to any existing RAID-0 file, use only on error

Suitable for striped files - add N parity per M data stripes (e.g. 16d+3p)

- Parity declustering avoids IO bottlenecks, CPU overhead of too many parities
- Should take failure domains into account (avoid data and parity on same OSS)
  - e.g. split 128-stripe file into 8x (16 data + 3 parity) with a total of 24 parity stripes

dat0	dat1	...	dat15	par0	par1	par2	dat16	dat17	...	dat31	par3	par4	par5	...
0MB	1MB	...	15M	p0.0	q0.0	r0.0	16M	17M	...	31M	p1.0	q1.0	r1.0	...
128	129	...	143	p0.1	q0.1	r0.1	144	145	...	159	p1.1	q1.1	r1.1	...
256	257	...	271	p0.2	q0.2	r0.2	272	273	...	287	p1.2	q1.2	r1.2	...

FLR Phase 4: Erasure Coded File Writes

Hard to keep data and parity in consistent during overwrite (RAID hole)

- Overwrite in place is fairly uncommon for most workloads
- Don't try to keep *parity* in sync during overwrite
- With Phase 1: mark parity component `STALE` during overwrite
  - Resync parity component when write is finished as with replica components
- With Phase 2: create and write temporary mirror replica instead of parity replica
  - Data age determined by whether allocated blocks exist in mirror component (FIEMAP)
  - Merge data from mirror to parity when write finished, skip holes in mirror (no new data)
  - Drop temporary mirror replica after merge is finished to save space