Lustre File Data Resiliency: Erasure Coding With Friends

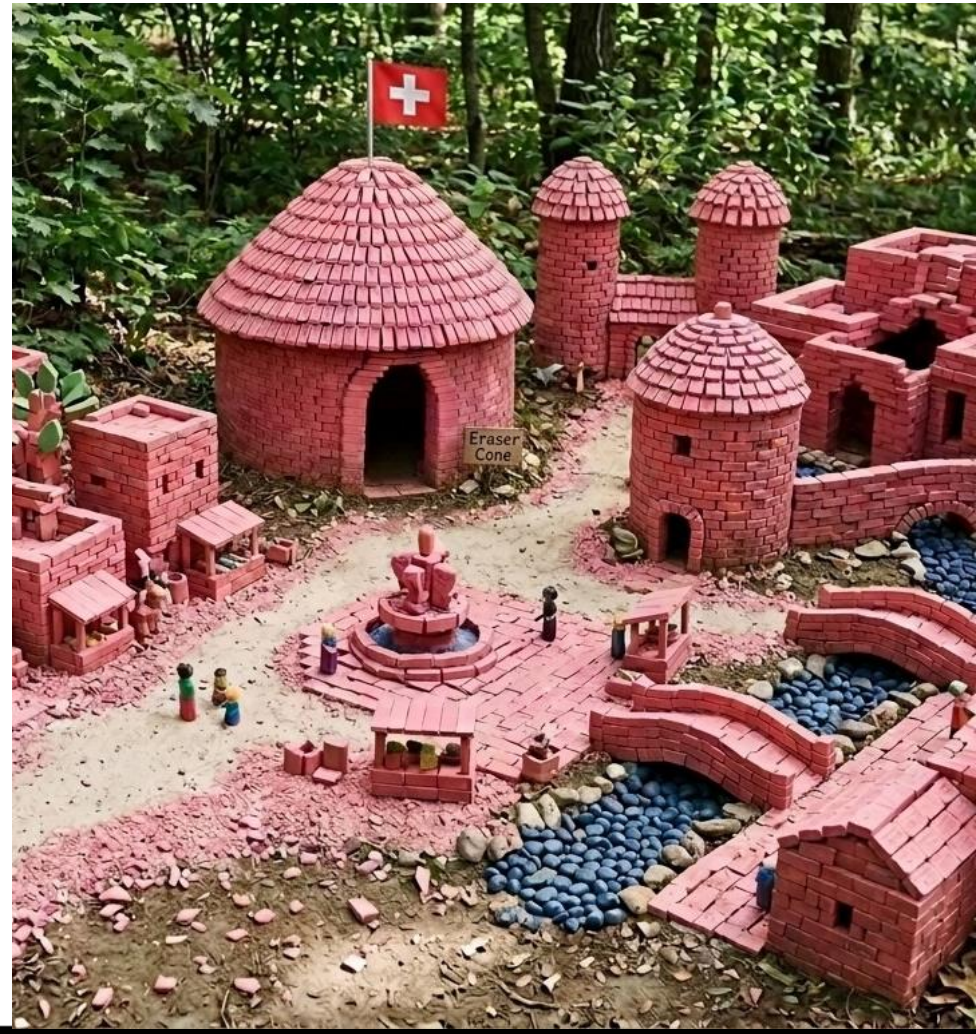
Patrick Farrell, The Lustre Collective

Lustre Erasure Coding – It Takes a Village

Started development over 8 years ago:

- 2017 – Intel – Zhenyu (Bobijam) Xu (LAD2018)
- 2021 – ORNL – Adam Disney (LUG2021)
- 2024 – ORNL (LAD2024)
  - James Simmons, Chris Brumgard
- 2025 – DDN/Whamcloud/TLC (LUG2026)
  - Marc Vef, Ronnie Sahlberg, Max Dilger
  - Patrick Farrell, Andreas Dilger, Hiroshi Nishida

Many thanks to everyone involved in bringing this important feature into the 2.18 release!



Lustre: Increased Scale Meets Hardware Limitations

- More devices = more frequent failures
- Extreme scale means hardware failures are routine
  - A single filesystem may have 10000 storage devices, 1000 servers, 4000 network cables/ports
  - Most of the nodes and storage are on OSTs, so they fail most frequently
- This has been partially mitigated in several ways …

  … but the cost of doing so keeps rising
  - OSTs have become larger over the years
    (100 TiB was originally a whole Lustre filesystem, now 100-1000 TiB *per OST*)
  - Added multiple network interfaces for better connectivity
  - Flash devices more reliable than spindles
- Heavy hardware requirements increases costs, increased downtimes… etc.
- Device scaling/resilience only gets you so far, and what about flakey networks?

Lustre: Data Redundancy

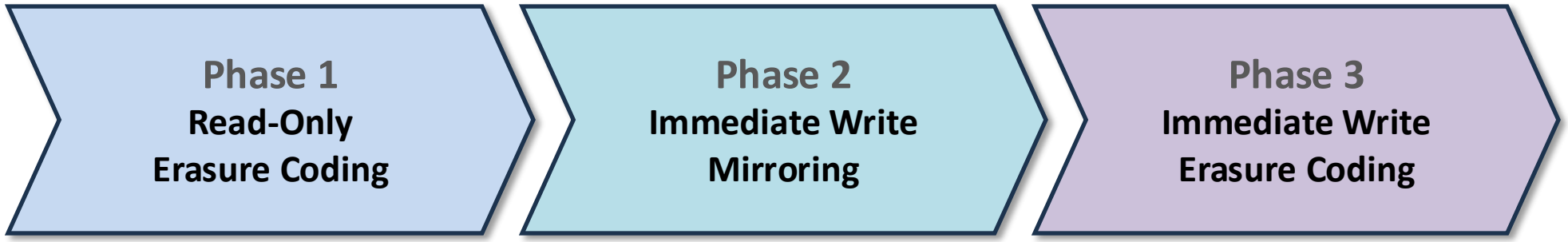
- Tolerate OST failure in software – break assumption that hardware is always available
- So, what does Lustre file-level redundancy look like?
  - Lose up to ‘m’ OST objects and still be able to access file data
- Lustre data layout defined on a *per-file level* using individual ***OST objects***
  - Individual file has a unique layout, usually with data split across several OST objects
  - Can configure overhead and resilience on a per-directory/per-file basis
- Mirroring (RAID-1)
  - Make m+1 full copies of the data, tolerate 'm' disk failures
- Erasure coding (RAID-4,5,6, generalized EC)
  - Use clever math to make 'm' redundant chunks, tolerate 'm' disk failures without full copies

Lustre Data Resiliency: Today

- File Level Replication (FLR): Lustre replication infrastructure
- Lustre currently has read-only (delayed) FLR mirroring:
  - Clients write to only *primary* mirror, all other mirrors are made *stale*
  - Data can be synced to secondary mirrors with `lfs mirror resync` command
  - Once a mirror is in sync, can be read from *any* mirror
  - Writing marks all but one primary mirror out of sync (stale again)
- Allows only delayed redundancy at high cost
  - Useful for offline tiering use-cases, mirroring idle data from SSD to HDD, critical files
  - Not space efficient, no immediate redundancy

Lustre Resiliency Project

Staged plan to improve data resiliency



- Read-only Erasure-Coding
- Delayed/Async parity resync
- Immediate benefits for users

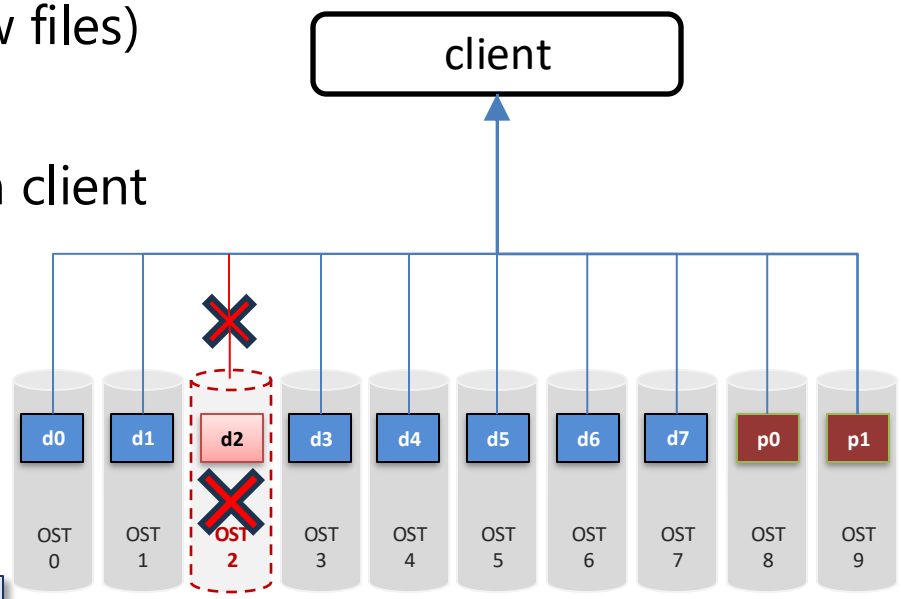
- Immediate/write-time fault tolerance
- Mirror space usage reduced after delayed EC parity resync

- Write-time, space-efficient fault tolerance

Each stage is useful independently, further increases data protection

Read-only Erasure Coding

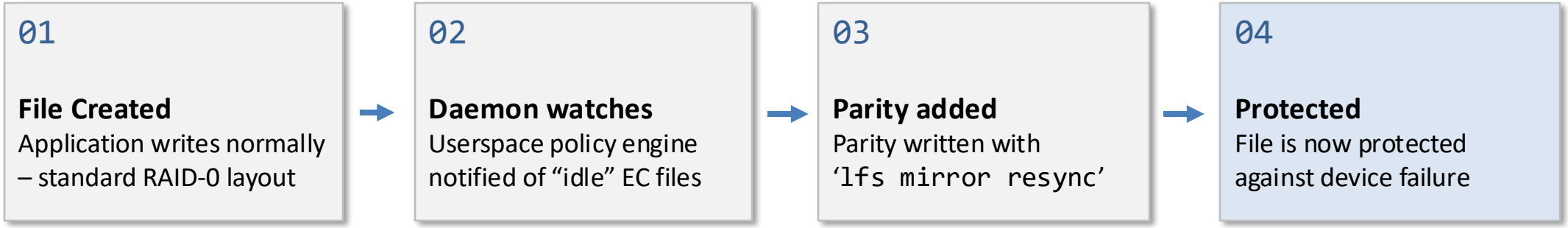
- Write striped file data normally, write redundant parity stripes later
- Once parity in place, reads can immediately bypass most OST and network failures
  - Data reconstructed on the fly by the client within seconds
- Works *like* read-only mirroring (for both old and new files)
  - Add **parity** after file is written instead of a full copy
- If OST unavailable, read data transparently rebuilt on client
  - Agnostic of storage, server, or network failures
- Reduces impact of OST failover
  - Reads continue within seconds of failure
- Far more space-efficient and robust than mirroring



8+2 EC: handle any 2 device failures, 25% bandwidth/space overhead
2-way mirror: handle 1 device failure, **100% write/space overhead**
3-way mirror: handle 2 device failures, **200% write/space overhead**

Read-only EC: how it flows

- Idle files with EC layout get protection eventually (within ~minutes)
- Not ideal for every application, but has real world uses and benefits
- Protects data for the 99.9% of its lifetime after write has completed



Use Case: Generative AI Training Data

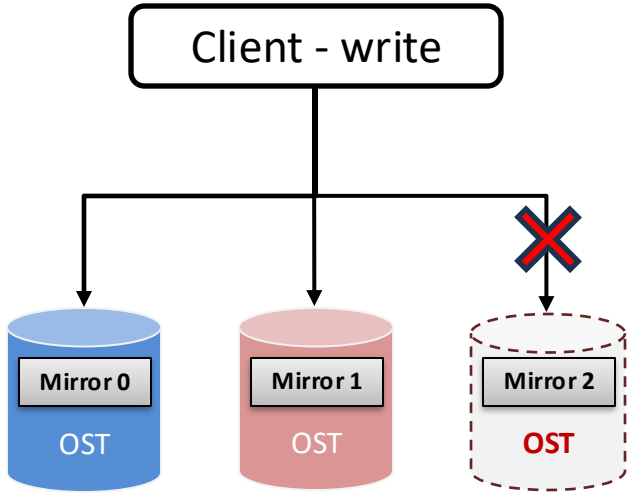
- AI training datasets are an ideal fit for read-only erasure coding
- Write once, read forever:
  - Training data is written once during dataset preparation
  - Read thousands of times during training runs across many nodes
  - Datasets only grow — existing data is never modified
- Read failure during a training run can force a costly restart
  - GPU hours are expensive and should be continually used
  - Multi-day training interrupted by storage failure wastes significant resources, even with checkpoints
- Read-only EC protects this data with minimal storage overhead
  - 8+2 EC: survive any 2 device failures with only 25% extra storage
  - 24+3 EC: survive any 3 device failures with **only 12% extra storage**
  - Compare to mirroring: **100% storage overhead for single-failure protection**

Read-Only EC: Limitations

- No immediate coverage for newly-written data
  - Need to opt-in pre-existing files to add EC mirror component
  - Single-stripe data components are still mirrored (at start of file, usually 5-10% of total capacity)
- No resilience for files which are continually modified
  - AI training checkpoints
    - Only safe **after** data written and file is idle for some time
    - Can't survive OST loss during write
  - Database files, other output files that are updated actively
    - No protection from delayed EC phase
- Next step: **Immediate write mirroring**

Immediate Write Mirroring (IWM)

- Clients send all writes to all file mirrors immediately
- Write failures handled transparently
  - As long as one mirror can complete a write, job continues
- Higher space overhead, but can be used for critical files
  - Checkpoints, databases, other commonly modified files
  - Also improves ***read*** bandwidth if files are frequently read
- Space overhead can be reclaimed after file becomes idle
  - **File can be mirrored at creation for immediate redundancy**
  - **Later migrated to read-only EC for space efficiency**



Immediate Write Mirroring: Flow

- Clients send all writes to all file mirrors immediately
  - Doubles write bandwidth used by the client
- Clients notify metadata server before they begin writing and after completion
  - Opening and closing a “write epoch” for each batch of writes
- Metadata server picks a primary mirror/“write leader” for this write phase
  - Metadata server marks non-primary mirrors "partially" stale during write (see design doc for details)
  - During write, mirrors are not always identical due to write ordering, network delays
  - Only primary mirror is readable during write to guarantee all clients see the same data
- After write, clients inform MDT of any errors. If none, MDT makes all mirrors not-stale.
- In case of error writing to a mirror, MDT marks that mirror as stale
  - Writes continue to update other non-stale mirror(s)
  - Stale mirrors repaired by full data copy from a good mirror via userspace resync tool, as before

Use Case Example: Full Gen-AI Training Runs

- AI *training* datasets are the ideal fit for read-only erasure coding
  … but training runs also generate ***checkpoints***

- Checkpoints save progress during a run (GPU and network failures are inevitable)
  - Written many times during every run to preserve current state
  - Should have redundancy during the write phase to ensure recoverability

- Solution: Immediate write mirroring for checkpoint, RO-EC for training data
  - Mirror checkpoint files when they are written (**not** all training data)
  - Provides fault tolerance for whole training run
  - **Not as efficient as EC, but checkpoints much smaller than training data**

- Immediate write mirroring also useful for KV Cache files

Immediate Write Mirroring

- Builds infrastructure for handling simultaneous writes to multiple destinations
  - Will also be used for immediate write erasure coding
- Design mostly complete, starting implementation planning: https://wiki.whamcloud.com/display/PUB/Immediate+Write+Mirroring+Design
- Will begin development **while read only EC is finalizing landing in 2.18**
  - Expect design completion shortly (hopefully in the next 3-4 weeks)
- Expected development time ~6-8 months (comparable to RO-EC)
- Add resources as development gets started
  - Depends on resources available, determining specific needs, ...

‘Immediate’ Write Erasure Coding

Erasure code data as it’s being written. Full fault tolerance throughout the process

End Goal

Full protection with EC efficiency

Leverages immediate mirroring infrastructure to update data and parity concurrently

vs. EC-Read-Only

Protection during write phase

No gap between write and parity – always protected

vs. Immediate Write Mirror

Space Efficiency and Lower Write Overhead

Avoid re-read of data when creating delayed EC/mirror

Immediate Erasure Coding: Outlook

Design

Design Outline Complete

Further design needs Immediate Write Mirroring to be completed

Dependencies

Builds on IWM

Will use IWM infrastructure to write parity at the same time as data

Timeline

~6-8 months after IWM is finished

About same development scope as IWM/RO-EC

Status

More to say soon

Detailed design begins once IWM is in motion

The Road Ahead

Progressive improvements that reduce impact of device failure
Increasing data protection with limited impact on capacity

- **Code-complete for 2.18**
- **Ongoing work for review, testing, patch landing**
- Value for read-heavy workloads with minimal space overhead

- Development starting
- Provides immediate protection (at higher space overhead)

- The end goal
- Immediate resilience at 25% overhead instead of 100%

Read-Only Erasure Coding: Technical Details

- More space-efficient than mirroring: 8+2 EC = 25% overhead vs 100% for a mirror
- All parity stored in separate FLR parity objects (RAID-4, **not** RAID-5/6)
  - *Parity naturally distributed* over OSTs by *different files* to avoid reconstruction hot spots
- Writes go to data mirror only; parity marked STALE (delayed parity)
- Resync tool computes parity from data using Intel ISA-L library
  - CPU optimized implementations (up to 20 GB/s/core) for x86, ARM, RISC-V
- If an OST fails, client reconstructs data from parity objects on the fly (degraded read)
  - Allocate new data/parity objects across all remaining OSTs for performance during rebuild
  - ***OST object rebuild needs free space in filesystem*** of at least one OST's space and inode usage!
- Backward compatible: older clients see unknown flag on parity mirror, skips it safely

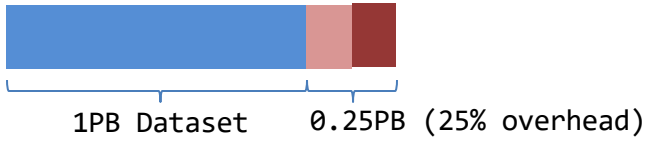
Space Efficiency: EC vs Mirroring

Scheme	Overhead	Failures
1 mirror	100%	1
2 mirrors	200%	2
4+1 EC	25%	1
4+2 EC	50%	2
8+2 EC	25%	2
16+2 EC	12%	2
16+3 EC	18%	3
24+3 EC	12%	3
32+3 EC	10%	3
32+4 EC	12%	4

Scenario: 1 PB Dataset

Mirroring – 2 failure protection

8+2 EC – 2 failure protection



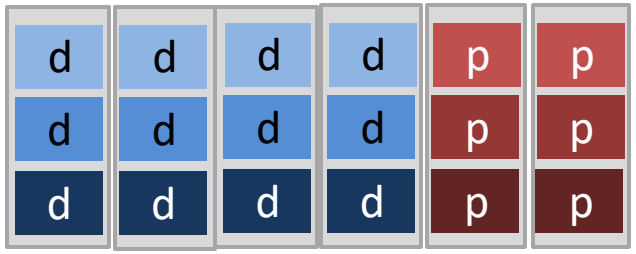
8+2 EC provides 1.75PB savings

At same protection level

EC: Why RAID-4?

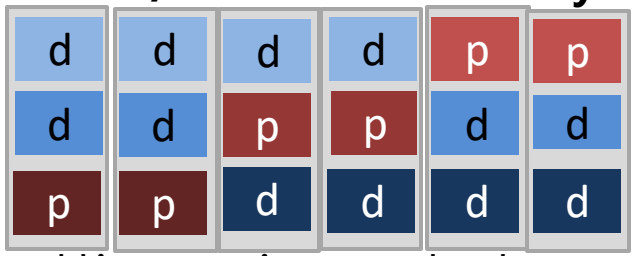
- RAID-4: Data and parity stripes each in separate OST objects
- Data uses existing RAID-0 data stripe layout as before EC
- Add new EC parity to striped files without rewriting any data
- Reuses FLR stale mirror handling: parity is an FLR mirror
- No new layout magic means old client/MDS can access file
  - Reuse standard `LOV_MAGIC_COMP_V1` / `LOV_MAGIC_COMP_V3`
- Parity component adds new layout flag for interoperation
  - `LOV_PATTERN_RAID0 | LOV_PATTERN_PARITY`
  - Old clients do not read/write PARITY mirror, only RAID-0 DATA mirror

RAID-4 – Dedicated Data and Parity Objects



Adding parity does not affect data

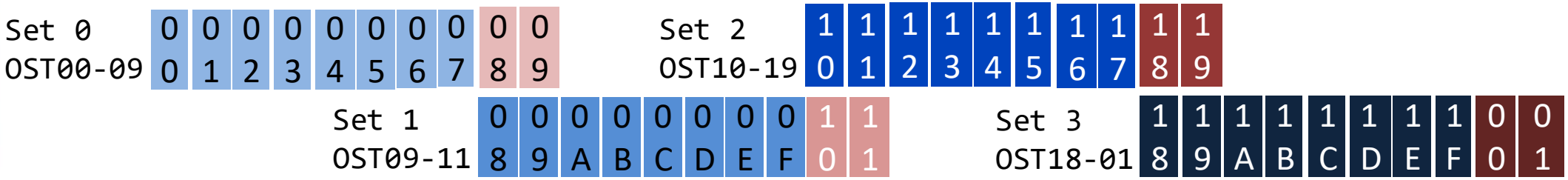
RAID-5/6 – Interleaved Parity Across Objects



Adding parity needs data re-write

```
# enable 8+2 EC on new 8-stripe file at time of creation
$ lfs setstripe -E -1 -c 8 --ec 8+2 /mnt/lustre/newfile
# enable 8+2 EC on existing 8-stripe file after creation
$ lfs mirror extend –N --ec 8+2 /mnt/lustre/oldfile
```

EC: RAID Sets

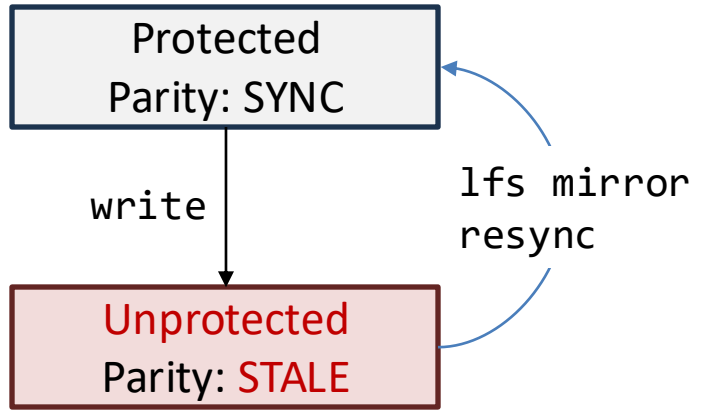


FILE: 32 data objects - 8d+2p EC - 4 EC RAID sets on 32 OSTs – 8 parity objects overlapping 8 data objects

- Widely-striped files divided into RAID sets, each an independent redundancy group
- Without RAID sets, parity rebuild would read all stripes at once
- RAID sets also solve the all-OSTs-have-object problem:
  - A file has a data stripe on every OST, no unused OSTs to hold parity stripes
  - Can find 2 OSTs to allocate parity objects for each 8-stripe RAID set not used in *that* RAID set
  - *Parity objects* can reuse OSTs of *data objects* in another RAID set without loss of redundancy
- Uneven sets handled by `ec_split_stripes()`:
  n_0 sets of k_0 stripes, n_1 sets of $k_1=k_0-1$
- Sparse files: RAID sets entirely in a hole use no parity storage

EC: Writes and Parity Lifecycle

- Writes go **ONLY to the primary data mirror** (standard RAID-0 striped IO)
- No bandwidth overhead during initial application write
- Parity mirror is automatically marked STALE on write (delayed parity)
- After write, parity is out of date until resync:
  - `lfs mirror resync` computes parity from data using ISA-L
  - Block-granular: uses `SEEK_DATA`/`SEEK_HOLE`, only resyncs regions with actual data
- ChangeLog records stale transitions so policy-engine can trigger resync without scanning
- `lfs mirror verify` validates stored parity matches computed parity



EC-RO Gap

Between write and resync the file has **no parity protection**

Addressed by immediate EC (Phase 3)

EC: Degraded Read Recovery

- Normal reads use only the primary RAID-0 data mirror objects (no parity needed)
- On OST read failure, client transparently switches to `CIT_EC_RD` IO type
  - Reads immediately fetch parity and reconstruct data without waiting for failed OST
- Two-layer IO model:
  - Outer IO: original VFS read request (application offset/count)
  - Inner IO: expanded to full RAID set (all available data + parity stripes)
- Parity pages are temporary VM pages, not cached (discarded after recovery)
- A stale parity mirror cannot be used for recovery (must resync first)
- Repair: `lfs migrate` creates new file, copies data via degraded read, deletes old file

EC: Design Decisions and Future

- Key design decisions:
  - RAID-set-aware OST object allocation preserves single-failure isolation within each RAID set
  - Backward compatible: older clients skip unknown PARITY flag, read/write data safely
  - On-disk format changes minimally affect existing file layouts:
    - `lcme_flags` adds `LCME_FL_PARITY`
    - `lcme_dstripe_count` (k), `lcme_cstripe_count` (m)
  - Standard limits k <= 32, m <= 4; expert mode (`--ec-expert`) k <= 255, m <= 15
  - Use highly-optimized Intel ISA-L EC implementation for reliability and performance

- Not yet supported:
  - No immediate parity computation on write (future work)
  - Single-stripe replacement not implemented yet (currently `lfs migrate/mirror` whole file)
  - Unable to write to a file with a failed data stripe until recovery or rebuild is finished
  - Resync depends on userspace-only tools, not fully automatic

Thank you! Questions?

Patrick Farrell <patrick@thelustrecollective.com>