
whamcloud

Internal technical seminar
May 29th 2012

Inode Iteration and OI Scrub

- Fan Yong

  Whamcloud, Inc.

  yong.fan@whamcloud.com

Agenda

- Background
- Requirements
- OI scrub
  - Basic mechanism
  - Checkpoint support
  - Trigger strategy
  - RPC service during OI scrub
- Inode iteration
  - Otable-based DT iteration APIs for up layer LFSCK
  - Rate control
- Userspace tools
- Tests

Background

- In Lustre-2.x, FID is the global unique identifier for the file/object
  - Independent from backend filesystem
- For osd-ldiskfs, Object Index (OI) files are used for FID ⇔ ino#/gen mapping
  1. The ino#/gen will be reallocated after restored from file-level backup
  2. Some OI file(s) may be corrupt/lost because of system crash
  3. Split/merge OI files for scalability
  4. OI files consistency routine check
- Only osd-ldiskfs for the contract
  - The first phase and the basic elements for the whole LFSCK

Requirements

- Online LFSCK
  - System is available during OI scrub
    - RPC w/o FID, or with new FID exported (by low layer) after the latest MDT mount up can be processed as normal, no need to wait.
    - RPC with old FID exported before the latest MDT mount up, e.g. replay or re-export through NFS, may fail directly or be blocked until related mapping is updated or OI scrub completed.
    - Performance may be affected, but correctness will not.
- Rate control
  - OI files consistency routine check (background OI scrub) should not impact other operations performance too much.
  - Speed limit can be adjusted during LFSCK running.

Requirements (cont'd)

- Controlled from userspace
  - LFSCK (OI scrub is contained) can be launched periodically or manually by user command.
  - LFSCK can be stopped by user command.
  - LFSCK real-time information, like status, progress (current position), speed, and so on, can be queried from userspace.
- Checkpoint support
  - Resumed LFSCK from the latest checkpoint
- General framework for LFSCK
  - Multiple components for the new LFSCK:
    OI scrub, MDT-OST consistency (layout, owner), DNE consistency
  - Shared inode iteration, rate control, userspace tools

How to rebuild OI files?

- FID in LMA
  - The FID is stored as part of the inode extended attributes, called as LMA (Lustre Metadata Attribute)
  - The FID in the inode LMA is always trusted
  - LMA will be preserved after restored from file-level backup
- Rebuild OI files with LMA
  - Locate OI mapping entry with the FID in LMA
  - Update OI mapping if unmatched
  - Insert new OI mapping if no-entry

Checkpoint support

- Checkpoint file on the device
  - New local file "OI_scrub" to trace OI scrub
  - "OI_scrub" is only visible inside osd-ldiskfs
- "OI_scrub" file structure
  - status: init, scanning, completed, failed, paused, crashed
  - flags: recreated, inconsistent, auto
  - latest checkpoint: for resuming from crash
  - statistics: file count (scanned/updated/failed), time
- Resume from latest checkpoint
  - Next start position will be the latest checkpoint position
  - "OI_scrub" is updated periodically (60 seconds)
  - If crash, at most one update cycle work may be lost

Trigger strategy

- Auto detect file-level backup/restore when mount
  - Old device UUID has been saved in "OI_scrub" before backup
  - New device UUID will be regenerated after restored
- Auto detect new created OI file(s) when mount
  - OI files count has been saved in "OI_scrub" when the first mount
- Auto check crashed OI scrub when mount
  - Status is "scanning" before OI scrub start
- Auto verify OI consistency during RPC process
  - Before exporting FID out of OSD
  - When lookup by FID
- Start/stop from userspace by force

Infrastructure for OI mapping

- Per-thread based single-entry cache
  - For current FID ⇔ ino#/gen mapping, whether related mapping in the OI file is correct or not, exist or not.
  - Filled by RPC service thread before exporting FID out of OSD:
    osd_ea_lookup_rec(), osd_it_ea_rec()
  - Accelerate OI lookup for subsequent FID-based operations.
- OI scrub high-priority inconsistent mappings list
  - For the right FID ⇔ ino#/gen mappings, if related mapping in the OI files are invalid.
  - Filled by RPC service thread when finds inconsistent OI entry:
    osd_ea_lookup_rec(), osd_it_ea_rec()
  - To guarantee subsequent FID-based operations (whether from the same thread or not) can find the right inode.
  - OI scrub will fix related entries in such list with high-priority.

lookup_by_FID with OI scrub

- Search the FID ⇔ ino#/gen mapping with the following order (to next step if formers failed):
  1. Current service thread OI mapping cache
  2. OI scrub high-priority inconsistent mappings list
  3. OI files
- Verify related mapping in the OI file with the FID in the inode LMA if comes to the 3rd step.
  - If inconsistent (for replay, re-export Lustre through NFS)
    - Trigger OI scrub if it has not run yet
    - Return "–EINPROGRESS" to client to notify the event
- How to process the "-EINPROGRESS" on client?
  - Retry as quota case does
  - Fail out directly (current behavior, may be adjusted in future)

Inode iteration

- Rebuilding OI files involves most of the objects on the device
  - osd-ldiskfs view: inode table based iteration is the most efficient way
    1. Scanning the inode table sequentially
    2. For the valid bit, get inode and the FID in LMA

       osd_scrub_next()
    3. Feed OI scrub with the right "FID ⇔ ino#/gen" mapping

       osd_scrub_check_update()
    4. Repeat above steps until the device is fully scanned
  - Inode read-ahead for more efficient disk I/O
    - Now, it is controlled by ldiskfs

      __ldiskfs_get_inode_loc()
    - Will consider to implement our own in LFSCK phase IV if needed

Inode iteration (cont'd)

- LFSCK components also fully scan the system
  - MDD view: namespace based scanning (traverse directory) is intuitive, but cannot guarantee full scanning because of rename.
  - Inode iteration is used to implement otable-based (object table based) DT iteration APIs which are exported by OSD to up layer LFSCK.

```
const struct dt_index_operations osd_otable_ops = {
        .dio_it = {
                .init           = osd_otable_it_init,
                .fini           = osd_otable_it_fini,
                .get            = osd_otable_it_get, /* specify iteration position */
                .next           = osd_otable_it_next,
                .key            = osd_otable_it_key,
                .key_size       = osd_otable_it_key_size,
                .rec            = osd_otable_it_rec, /* return FID */
                .load           = osd_otable_it_load,
        }
};
```

OI scrub modes

- Urgent OI scrub
  - Recreated: OI files are removed/recreated
  - Inconsistent: restored from file-level backup
  - Auto: inconsistency detected during RPC process
- Non-urgent (background) OI scrub
  - OI consistency routine check
  - Run background automatically when other LFSCK

Rate control

- Under urgent mode, OI files should be rebuilt/ updated as soon as possible, no speed limit
  - Try the best to guarantee system fully available
- For background OI scrub, to reduce performance impact on others, need rate control
  - Controlled by otable-based DT iteration rate
  - Main LFSCK engine invokes otable-based DT iteration
  - Prefetch window between OI scrub and up layer LFSCK otable-based DT iterator (1024 inodes)
  - Specified when start LFSCK from userspace
  - Adjustable during LFSCK running

Userspace tools

- Start LFSCK by command

lctl lfsck_start <-M | --device MDT_device>
[-e | --error error_handle] [-h | --help]
[-m | --method iteration_method]
[-n | --dryrun switch] [-r | --reset]
[-s | --speed speed_limit]
[-t | --type lfsck_type[,lfsck_type...]]

OPTIONS:

-M: The MDT device to start LFSCK on.

-e: Error handle, 'continue'(default) or 'abort'.

-h: Help information.

-m: Method for scanning the MDT device. 'otable' (otable-based iteration, default), 'namespace' (not support yet), or others (in future).

-n: Check without modification. 'off'(default) or 'on'.

-r: Reset scanning start position to the device beginning.

-s: How many items can be scanned at most per second. '0' means no limit (default).

-t: The LFSCK type(s) to be started.

Userspace tools (cont'd)

- Stop LFSCK by command

  lctl lfsck_stop <-M | --device MDT_device> [-h | --help]

  OPTIONS:

  -M: The MDT device to stop LFSCK on.

  -h: Help information.

- Query LFSCK information by command
  - Every LFSCK component has its own special lproc interface
  - For OI scrub:

    lctl get_param -n osd-ldiskfs.${MDTDEV}.oi_scrub

- Adjust speed limit during LFSCK running

  lctl set_param -n mdd.${MDTDEV}.lfsck_speed_limit=N

  Options:

  o: no speed limit.

  Others: scan at most N objects per second.

New mount options – "noscrub"

- Do not trigger OI scrub automatically
  - NOT start/resume OI scrub automatically when MDT mounts, even though some OI inconsistency is detected.
  - Prevent OI scrub to be triggered automatically if some bad OI entry is found during system service.
- Ignore it if trigger OI scrub with user command
- Can be overwritten by lproc interface after MDT mount up

lctl set_param -n osd-ldiskfs.${MDTDEV}.auto_scrub=N

OPTIONS:

o: cannot trigger OI scrub automatically.

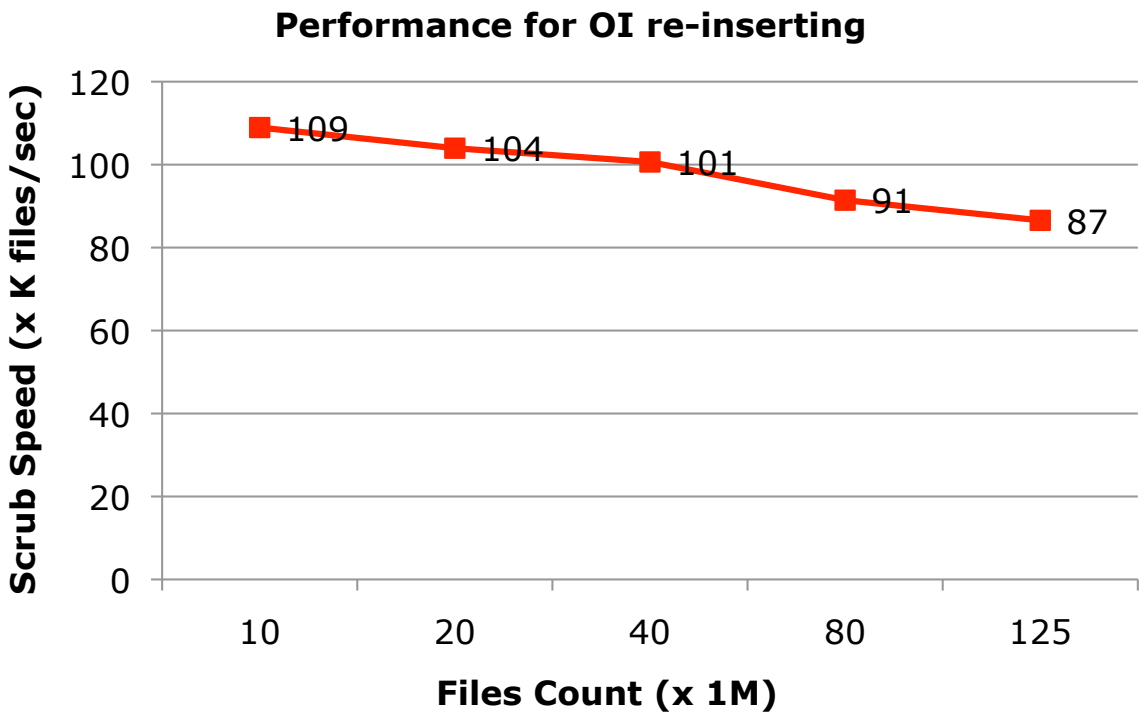
Others: can trigger OI scrub automatically when needed.

Tests

- Hardware: fat-intel-2 on Toro
  - CPU: 2 x Intel® Xeon® X5650 2.67GHz, Six-core Processor, 2-HT for each core
  - RAM: 24GB DDR3 1333MHz
  - Disk: 250GB SATAII Enterprise Hard Drive
  - Journal: external journal on 8GB SSD
- Configuration
  - Single MDT w/o OST and w/o client
  - Use 64 OI files on the MDT by default
- Method
  - echo_client drives the MDT directly with 0-striped objects created

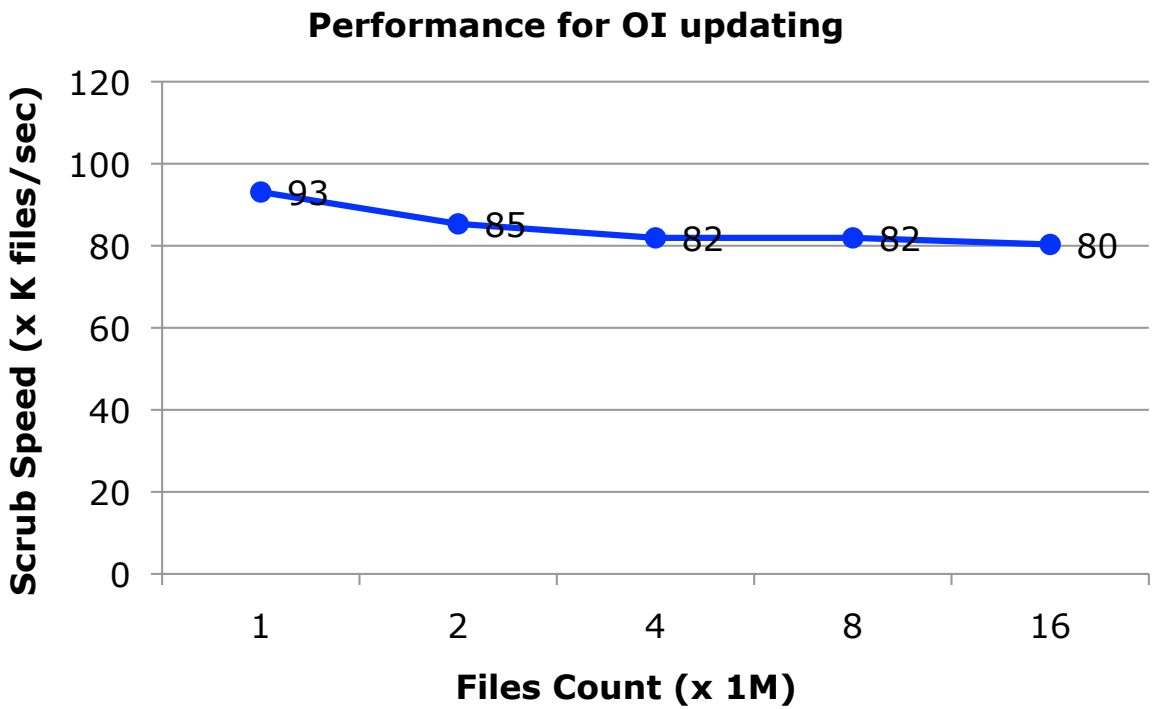
Test1: scrub speed for OI files remove/recreate

- Method
  - Re-insert OI mapping entries after all OI files removed/recreated



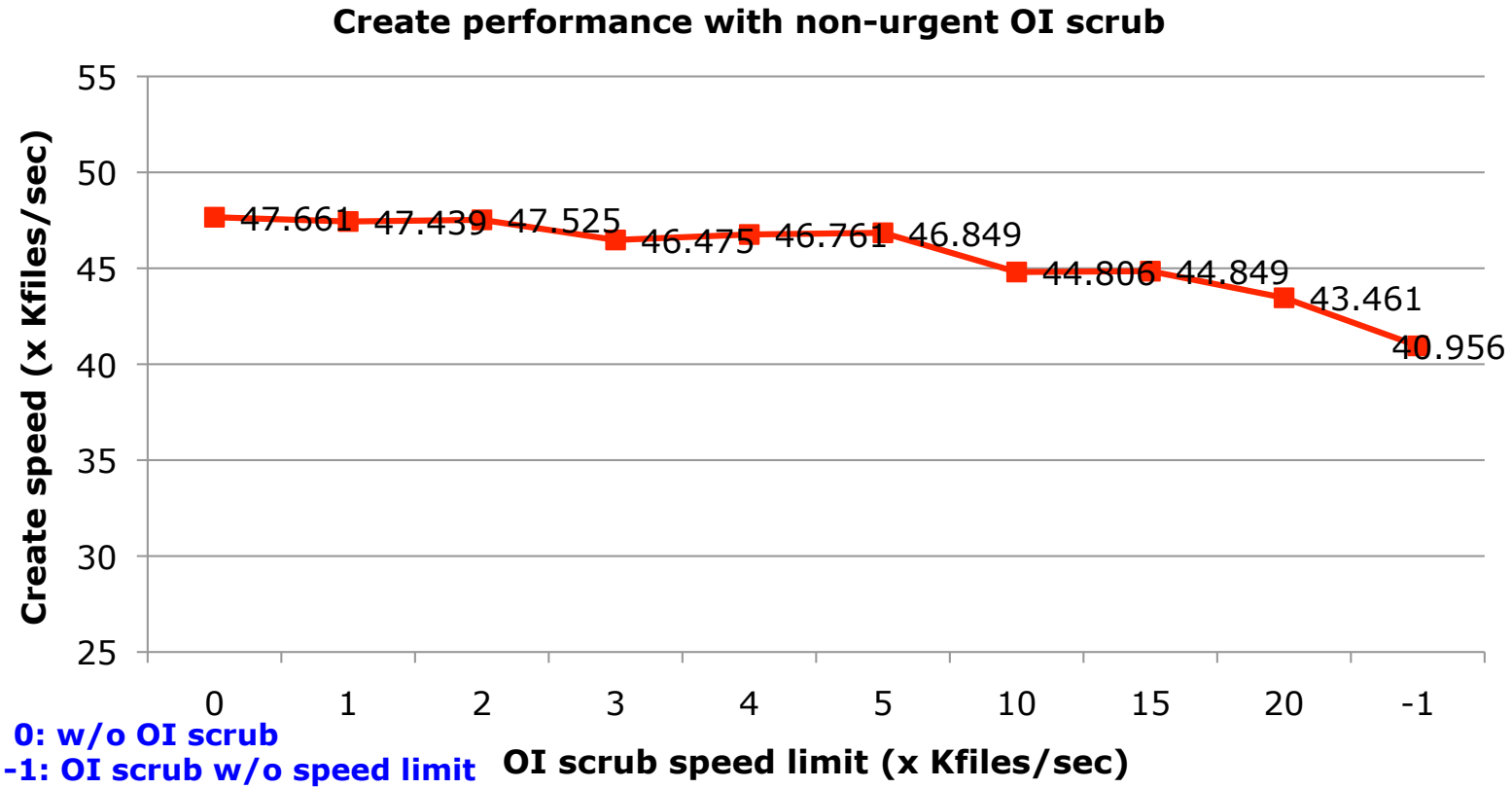
Test2: scrub speed for MDT backup/restore

- Method
  - Update OI mapping entries after MDT restored from file-level backup



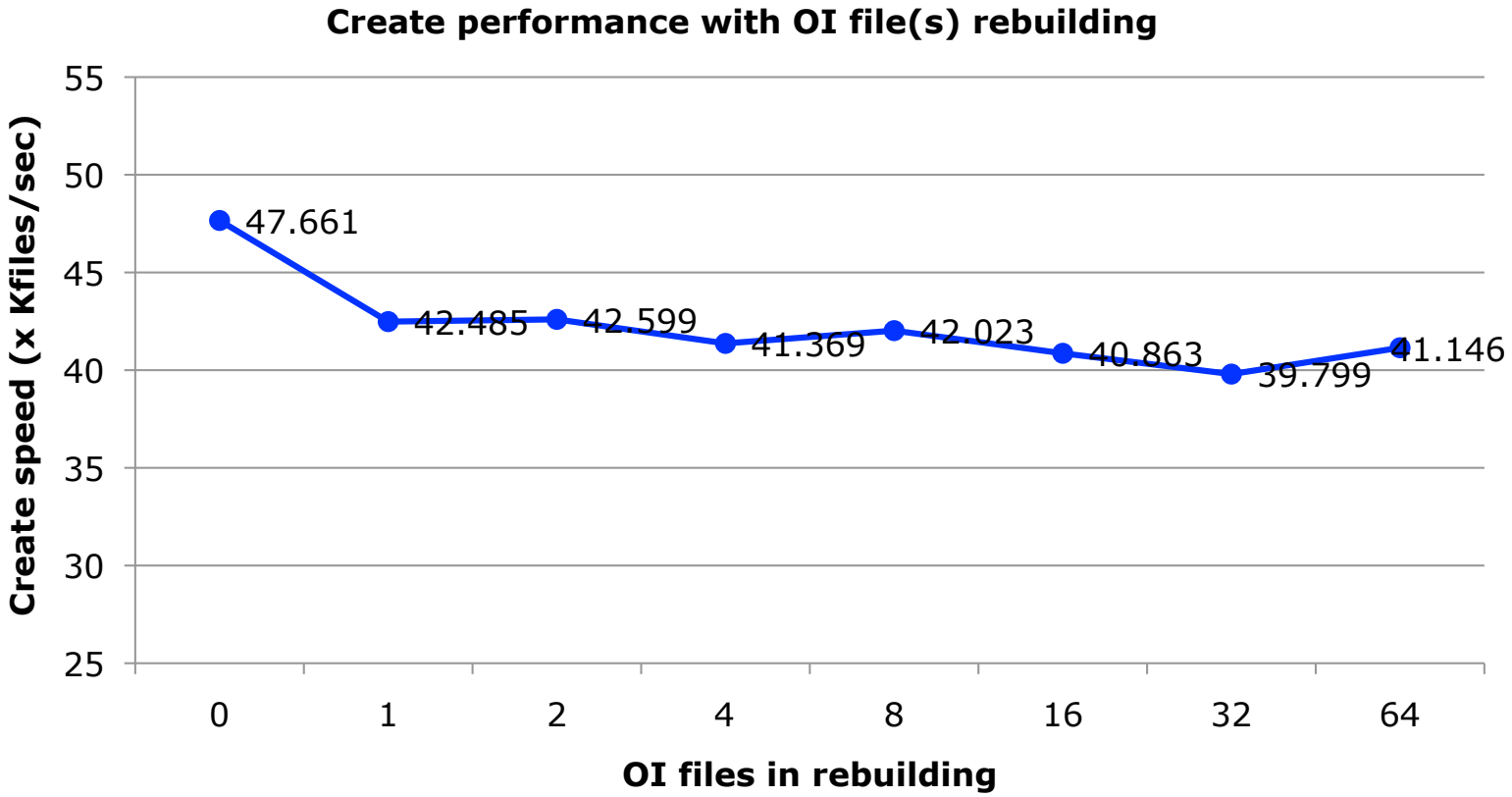
Test3: performance impact for create with non-urgent (background) OI scrub

- Create with OI scrub run background with kinds of speed limit (full speed is about 20K/sec). The create is driven by echo_client with 64 threads for 2,560,000 files under per-thread based directories.



Test4: performance impact for create with urgent OI scrub

- Create with OI scrub rebuilding different numbers of OI files. The create is driven by echo_client with 64 threads for 2,560,000 files under per-thread based directories.



Conclusions

- Rebuilding OI files from empty state is faster than updating the existing OI files.
- Within 25% of the full speed of background OI Scrub, the performance impacts for create is less than 3%, almost can be ignored.
- Under urgent OI scrub mode, the performance impacts for create is about 15%. The tendency between performance impact and OI files count in rebuilding is not distinct.

Thank You

- Fan Yong

  Whamcloud, Inc.

  yong.fan@whamcloud.com