Data On MDT High Level Design

Introduction

The Data-on-MDT project aims to achieve good performance for small file IO. The motivations for this and current behaviour of small file IO are described in Data-on-MDT Solution Architecture document.

This high-level design document provides more details about approach to implement this feature in the Lustre* file system.

Implementation Requirements

Create new layout for Data-on-MDT

While Layout Enhancement project is responsible for a new layout, we must add appropriate changes in LOV to properly recognize the new layout and use LMV/MDC `cl_object` interface to work with data. LOV should understand and properly handle also maximum data size for the new layout. Another part of related work is to change lfs tool so it will be able to set new layout on directory as default.

CLIO should be able to work with MDT

Make MDC devices part of CLIO

MDC devices must become `cl_device` and part of CLIO stack and be used by LOV for IO if new layout. Some unification is possible between OSC and MDC methods.

> That code is orthogonal to the current MDC and is placed there for simpler access to import and request-related code. This code will become eventually the generic client that will be used by MDC and OSC.

MDT support for IO requests

IO request handlers

Unified Target made this possible, but specific handlers do not currently exist for the MDT. They will be added as part of this work.

LOD must understand new layout

LOD must be aware of a new layout and handle it properly bypassing IO methods and other operations to the local storage instead of OST

Lustre tools changes

The `lfs setstripe` command must be extended to recognize new layout setup and set maximum size for it.

Functional Specification

New DOM layout

Wire/disk changes

The `lov_stripe_md` (lsm) stores information about DOM layout. The `lw_pattern` has specific flag `LOV_PATTERN_DOM` set, the `lw_stripe_size` contains maximum data size value for DOM, the `lw_stripe_count` is 0, no `lsm_oinfo` is allocated. This does not change wire protocol because `LOV_PATTERN_DOM` replaces existing `LOV_PATTERN_FIRST`.

In-memory structures

The LOV implements new interface for new `LLT_DOM` layout type. This set of methods are closely related with `LLT_RAID0` interface and some may be reused.

Upon new object allocation the LOV does the following:

- `lov_layout_type()` checks `lsm` and returns related type;
- `lov_dispatch()` points to proper interface for this type;

so the proper stack of objects is built with using sub-objects points to MDC.

```
union lov_layout_state {
  struct lov_layout_raid0 {
   ...
  } raid0;
  struct lov_layout_dom {
   struct lovsub_object *lo_dom;
  } dom;
 }
```

The `lov_layout_dom` refers just to single object below LOV.

New LOV subdevices

For DOM we need pass IO through the MDC device, therefore LOV should have such target similar with other targets pointing to OSCs. With DNE in mind we should be able to query FLDB by FID to get proper MDC device according. The LOV implements own `lov_fld_lookup()` for that and may need to setup also own fld client OR use one from LMV.

```
struct lov_device {
    ...

 /* Data-on-MDT devices*/
 __u32         ld_md_tgts_nr;
 struct lovsub_device **ld_md_tgts;
 struct lu_client_fld  *ld_fld;
};
```

Adding MDC

MDC devices are added in client LOV config log like them are added to the LMV. Upon new MDC addition the device stack may be not yet ready and must be saved to be added later in the `lov_device_init()`. The LOV device has new arrays of MDC targets `ld_device:ld_md_tgts[]` with the same type `lovsub_device` as used for OSC but with different set of methods.

Deleting MDC

Unlike OSC the MDC might be cleaned up before LOV because they are controlled still by MD stack of devices. The problem can be solved by taking extra reference for MDC device at LOV or better by notification of LOV about MDC is going to shutdown.

LOV `lovdom_device`

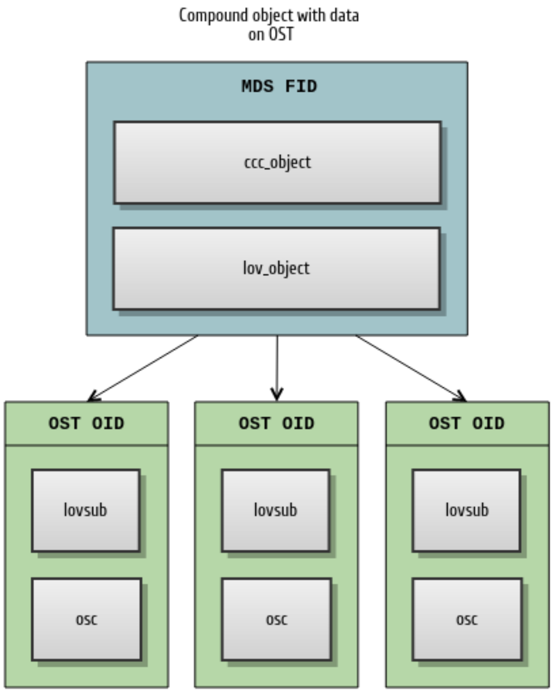
The `lovdom` device replaces lovsub device for OSC. New set of methods is introduced to handle LOV-MDC interactions.

```
static const struct lu_device_operations lovdom_lu_ops = {
 .ldo_object_alloc      = lovdom_object_alloc,
 .ldo_process_config    = NULL,
 .ldo_recovery_complete = NULL
};

static const struct cl_device_operations lovdom_cl_ops = {
        .cdo_req_init = lovsub_req_init
};


static struct lu_device *lovdom_device_alloc(const struct lu_env *env,
          struct lu_device_type *t,
          struct lustre_cfg *cfg)
{
 struct lu_device *d;
 struct lovsub_device *lsd;
 OBD_ALLOC_PTR(lsd);
 if (lsd != NULL) {
  int result;
  result = cl_device_init(&lsd->acid_cl, t);
  if (result == 0) {
   d = lovsub2lu_dev(lsd);
   d->ld_ops = &lovdom_lu_ops;
   lsd->acid_cl.cd_ops = &lovdom_cl_ops;
  } else {
   d = ERR_PTR(result);
  }
 } else {
  d = ERR_PTR(-ENOMEM);
 }
 return d;
}
static const struct lu_device_type_operations lovdom_device_type_ops = {
 .ldto_device_alloc = lovdom_device_alloc,
 .ldto_device_free = lovsub_device_free,
 .ldto_device_init = lovsub_device_init,
 .ldto_device_fini = lovsub_device_fini
};
```

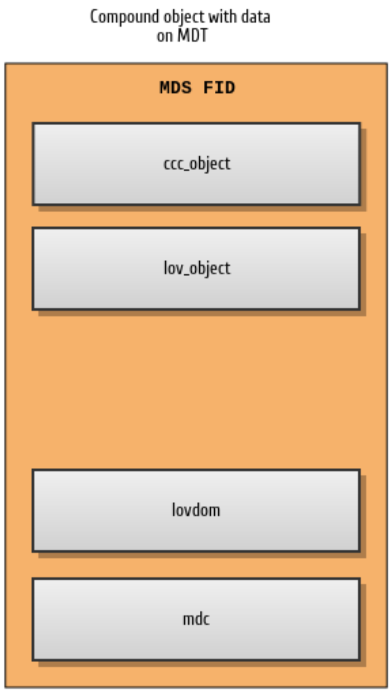
The `lovdom` device is much simpler than `lovsub_device` because it is not a top device of new sub-stack but part of the stack. That is why device alloc is different but at the same time we reuse other `lovsub` methods.

DOM object layering

DOM object is referred by single FID of MDS object and has no stripes, so it is presented by plain stack ccc->lov->lovdom[mds_number]->mdc referred by FID of MDS object.



Hierarchy model with upper object identified by MDS FID and set of bottom objects identified by OSD object IDs and managed by LOV.



Plain object identified by MDS FID

Considering there is always single stripe, the lov+lovdom layers are trivial and have almost no extra functions/checks. All real code goes to the MDC layer.

Manage max_stripe_size

LOV specific code is max_stripe_size boundary check and code to start migration upon file growing.

The `cl_io_operations:cio_io_iter_init()` method for `lovdom` layer checks that operation is not out of max_stripe_size boundary. The Phase I action is just -ENOSPC return code, the Phase II starts migration at this point.

MDC

MDC CLIO methods

MDC fully reuses OSC CLIO methods with several exceptions, e.g. the device initialization methods will be different as it must call `mdc_setup()` instead of `osc_setup()`. That means we can separate that code from OSC and make it generic.

Locking

All IO for DOM object is done under PW LAYOUT lock. It protects whole data on MDT exclusively. The llite is responsible for that and MDC CLIO locks operatoins are all lockless to avoid real lock enqueueing.

Glimpse

With data on MDT the GETATTR request from MDT returns both `OBD_MD_FLSIZE|OBD_MD_FLBLOCKS` along with other attributes. Therefore glimpse is disabled by setting inode flag `LLIF_MDS_SIZE_LOCK` for DOM objects.

MDT changes

IO request handlers

New set of handler is added at MDT, they are needed to handle punch/read/write operations.

```
static struct tgt_handler mdt_tgt_handlers[] = {
 ...
 TGT_MDT_HDL(HABEO_CORPUS| HABEO_REFERO, OST_BRW_READ, tgt_brw_read),
 TGT_MDT_HDL(HABEO_CORPUS| MUTABOR, OST_BRW_WRITE, tgt_brw_write),
 TGT_MDT_HDL(HABEO_CORPUS| HABEO_REFERO | MUTABOR, OST_PUNCH, mdt_punch_hdl),
}
```

IO object operations

New operations are added to the MDT and sends IO requests directly to the OSD bypassing MDD and LOD. Unified Target uses OBD interface for IO methods, so MDD OBD operations are extended with the following:

```
mdt_obd_ops = {
 ...
 .o_preprw  = mdt_obd_preprw,
 .o_commitrw  = mdt_obd_commitrw,
}
```

DOM support in LOD

LOD understand new DOM layout by checking lsm pattern with `lsm_is_dom(lsm)` helper function. It must not use any OSP subdevices for such layout but only local OSD target. All assertions and check on pattern should be extended from "only RAID0 is supported" to "RAID0 AND DOM are supported".

MDT stack doesn't filter BLOCKS | SIZE attributes from DOM object and return them to the client.

Quota on MDT

As a first implementation, a user limit the DOM blocks can be provided without any code changes: for instance, if user want to set 1GB limit for DOM and the maximum size of each file on MDT is 1M, then user can just set the inode limit to 1024. This approach is in conflict with a real inode limit but provides a initial path to quota control.

A more robust quota for Data on MDT can be executed in two phases:

Phase I: Support block quota on MDTs, and MDTs share same block limit with OSTs;

In current quota framework, each MDT has only one MD qsd_instance associated with it's OSD device

```
struct osd_device {
        ...
  /* quota slave instance */
  struct qsd_instance      *od_quota_slave;
};
```

This will be expanded to a list, and two qsd_instance will be linked in the list for each MDT: one is MD for metadata operations, the other is DT for data operations. Write on MDT will use the DT qsd_instance to enforce block quota, and metadata operations will have to enforce both block and inode quota with these two qsd_instances. For ZFS OSD, once block quota enabled on MDT, the approach of estimating inode accounting by used blocks will not work. Using our own ZAP to track inode usage will be the only option. Ideally, OpenZFS will support inode accounting directly.

Phase II: Support different block limits for DOM and OSTs;

Because space on a MDT is often limited compared to a OST, an administrator may want to set more restricted block limit to MDTs rather than sharing same limits with OSTs. To support private limits for DOM, following changes are required:

- Introduce a pre-defined DT pool for DOM to manage the block limit for all MDTs;
- Add an additional qsd_instance associated with each MDT. This acquires limits from the DOM pool; (some code could probably needs be revised to support non-default pool ID)
- Enhance the quota utilities to set and show quota limits for specified pool; (packing pool ID in quota request requires client to server protocol changes)
- Write on MDT must enforce two block quotas: the default DT quota and the DOM quota;

Grants on MDT

Grant support on MDT require major MDT stack changes because we need to account all changes including metadata operations with directories, extended attributes and llogs. Initial grants support will be implemented as MDT target support for grants basically. It is able to report grants and declare their support during connection, but returns zero values (means IO to be sync) or some rough estimated values. Further work will be done in accounting all operations with data and report real values. This can be done during declaration phase of transaction.

Operations to take into account:

- directory create/destroy
- EA add/delete
- llog accounting (changelogs mostly)
- Write/truncate of DOM objects
- DOM object destroy

Lustre tools changes

LFS setstripe

LFS tool introduces new option '`--stripe-mdt | -M`'. It means DOM layout and sets pattern to `LOV_PATTERN_DOM`, stripe counts to 0. Option `--stripe-size` used with `--stripe-mdt` will set DOM maxsize. Option `--stripe-count` will cause error if not 0. Pool name cannot be set also for this layout.

Migration

Migration happens during IO operation and consist of several steps:

1. determine a migration event
2. create a file with new layout
3. read data from old file and write it to the new one
4. swap layout between files
5. continue current operation

That means the migration manager should be done in llite in context of current operation or be a separate thread to avoid possible conflict which CLIO thread infos for second object.

Migration start

LOV determines if migration is needed. Migration is evaluated when ongoing WRITE request exceeds the `max_stripe_size` value of DOM object. The `cl_io_operations:cio_io_iter_init()` implementation for `lovdom` layer checks if migration is needed.

Layout change

New object is created with RAID0 layout, OST objects are taken from pre-allocated pool as usual during file creation on MDT. When data is moved from one file to another the layout is switched. All operations are done under PW `LAYOUT_LOCK` naturally so no additional locking is needed.

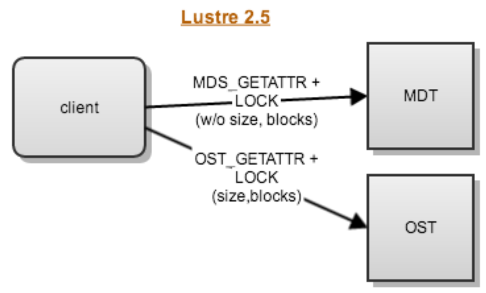
Data migration from MDT to OST

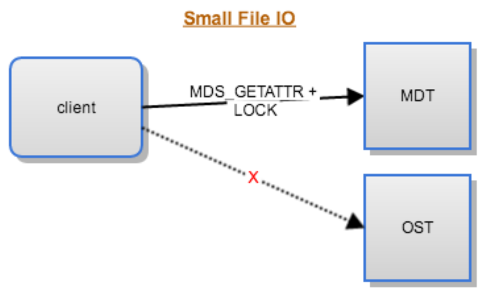
The whole data of DOM file is read from MDT by client and flushed back to the new RAID0 file. One possible optimization is to re-assign pages to the new file by flushing them from CLIO. This optimization is beyond the scope of this project.

Logic Specification

MDS_GETATTR logic

Optimization is to avoid glimpse requests by returning back SIZE and BLOCKS attributes.

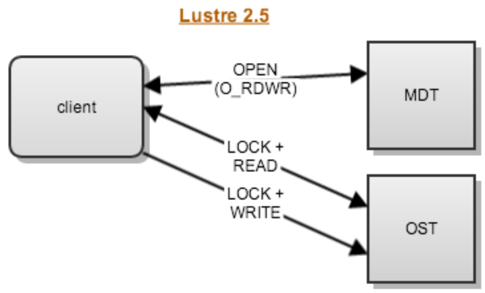

Lustre 2.5
client
MDS_GETATTR + LOCK (w/o size, blocks)
MDT
OST_GETATTR + LOCK (size,blocks)
OST

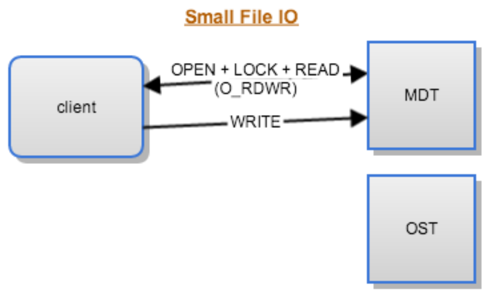


- MDS_GETATTR is enqueued as usual but with PR LAYOUT lock
- MDT returns BLOCKS and SIZE attributes as valid
- llite sets flag to indicate that glimpse is not needed (already exists for SOM)

MDS_OPEN logic and optimizations

OPEN with O_RDWR

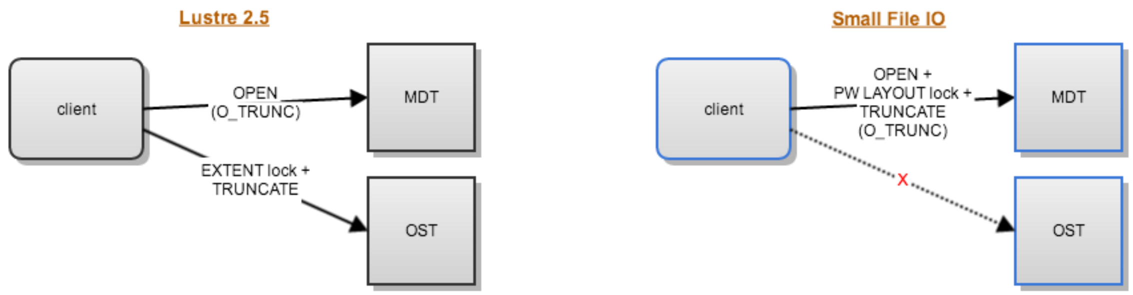




This optimization might help with partial page updates, when client have to read page first, adds new data in it and flush it back finally.

- OPEN is enqueued with PW LAYOUT lock on object
- MDT returns partial page back if fits into EA buffer
- client update partial page and flush pages

OPEN with O_TRUNC

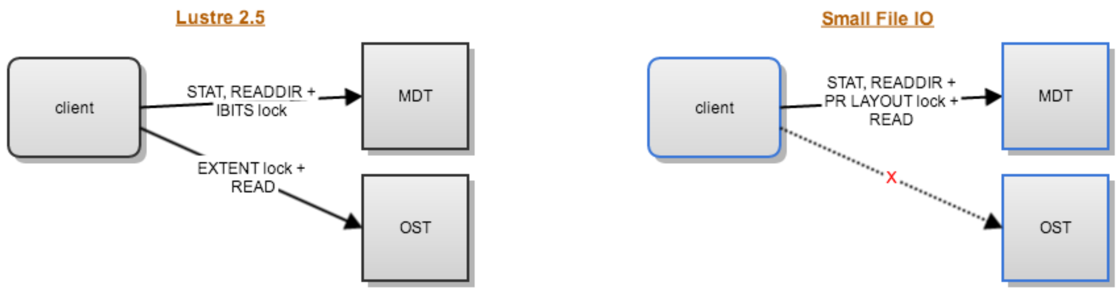


This optimization can truncate file during open if O_TRUNC flag is set.

- OPEN is invoked with PW LAYOUT lock
- MDS checks `O_TRUNC` is set and truncate file
- client gets reply with updated attributes

Data Readahead logic

All readahead optimizations are based on fact that it is possible to return some data from small file in EA buffer which is quite big now.



Typical case is stat():

- MDS_GETATTR is enqueued as usual
- MDT returns basic attributes and checks amount of free space in EA reply buffer.
- MDT reads data from file to EA buffer
- client fill pages with data from EA buffer

Tests Specification

Sanity tests

The set of tests to make sure the feature works as expected

test_1a - file with DOM layout

1. create file with `lfs setstripe -M`, check its LOV EA attribute has `LOV_PATTERN_DOM` pattern and has no stripes on OST
2. write to the file, checks data is written and valid, checks file attributes are valid, checks fs free blocks is decreased according with file size.
3. delete file, checks free blocks are returned back

test_1b - DOM layout as default directory layout

1. set new default layout on directory with `lfs setstripe -M` and create file in it.
2. all steps from test_1a

test_2a - check size limit and migration

1. create file with DOM layout
2. write more that max_stripe_size to the file
   a. Phase I: checks that `-ENOSPC` error is returned
   b. Phase II: check that migration occured
3. (only Phase II below)
4. checks file layout is changed to filesystem default layout and has stripes on OSTs
5. checks file size is valid and fs free blocks data is valid as well

Sanityn tests

test_1{a,...} - parallel access to the small files

1. open DOM file from client1 in different modes and pause it.
2. perform various operations on file from client2
3. make sure client2 is waiting client1 to finish first

Recovery tests

1. fail MDS while performing operations like create/open/write/destroy on DOM file
2. make sure operations are replayed back and finished as expected

Functional tests

Check all cases we optimize

Stat

1. create files with DOM layout vs 2-stripes layout
2. fills them with data
3. perform `ls -l` on them
4. Output results for both cases

Open with O_TRUNC

1. create files with DOM and default layout
2. fills them with data
3. perform open with `O_TRUNC`
4. output results

Write at the end of file with partial page

1. create many small files and fills with data so last page is filled partially
2. performs write to the end of file
3. Output results for DOM file vs normal file

Readahead of small files

1. create small files with DOM and normal layout
2. fills them with data
3. perform `grep` on them
4. output results

Generic performance tests

mdsrate

Regular MD operations should benefit from Data-on-MDT because there are no OST requests, basically only stat should benefit noticeable because open uses precreated objects and destroy is not blocked but OST objects destroy.

use mdsrate to perform the following operations:

1. file creation with single stripe vs DOM
2. stat
3. unlink
4. output results for both cases

> **files with data**
> The mdsrate tool must be modified to create DoM files with amount of data to test DoM files performance in common case.

FIO

Check generic IO performance of small files

1. run FIO with DoM files vs normal files
2. output results

postmark

The Postmark utility is designed to emulate applications such as software development, email, newsgroup servers and Web applications. It is an industry-standard benchmark for small file and metadata-intensive workloads.

1. Run postmark over DoM striped directory and default one.
2. Output results