Proposal for I/O Performance Improvement

Introduction

In this document, I will come up with an idea to change the architecture of Client side I/O stack to make it more performant for write. I have done some work to improve single thread writing performance in LU-3321, but it only benefits from I/O in large size.

In LU-3321, I discovered that one of major overhead in client I/O stack is to submit pages from LLITE layer to OSC layer. In order to overcome that issue, I added a list in the LLITE layer to accumulate dirty pages, and then at the end of the write, those dirty pages will be submitted in batch via the newly added interface `cl_io_commit_async()`. That was a significant improvement for write with large IO size. Small writes still have to suffer the performance penalty due to the overhead of `cl_io` initialization and page submission at the end of each write.

This document will describe a solution to reduce the overhead of I/O stack. The idea is to reserve whatever resources are needed to write pages to the LLITE layer. Therefore LLITE layer can make a decision by its own if it can cache the dirty pages in memory. A set of new mechanism will be invented to make things work correctly.

Use Cases

Write I/O Performance Improvement

This work is intended to improve write performance for both small and big I/O size. After this work is done, if the write pattern is sequential, small I/O performance is expected to see similar performance numbers as large I/O.

Partial Page Write

Partial page write for large I/O is usually not an issue because it's still pretty good even considering the overhead of reading a few pages from OST. When the I/O size is small, the reading overhead will become significant. This design will also solve those pathological cases of non-page-aligned IO size.

Write Container

I would propose to use the concept of write container to reserve resources from the OSC to LLITE layer so that LLITE can decide if it can cache dirty pages on its own discretion. In the initialization phase of IO, LLITE is going to transfer enough information such as type of IO, the extent of the object it intends to write, in the layers of I/O stack. This is actually a very good opportunity for the OSC to create a container, and put whatever resources it needs to cache dirty pages in the container and transfer it back to LLITE layer. The resources that need to dirty pages usually include object extents, LDLM lock, dirty pages accounting, space grant from OST, LRU slots of pages, and perhaps the available extent between existing osc_extents.

When the OSC creates a container, it has to take the current available resource into account. For example, it shouldn't reserve half of grant space to one container because concurrent I/O from other threads will drain the resource quickly. After LLITE gets a container, it can write dirty pages without talking to OSC. Full containers should be submitted by LLITE immediately, so that OSC can create osc_extent and add them into I/O engine.

The owner of write containers are OSCs, and OSCs maintain a list of containers that have been reserved. When the resource is in shortage at OSC, OSC should revoke the containers by traversing the list and invoke an callback provided by LLITE. On receiving the revocation, the LLITE should submit the container right away if it's not being used. Otherwise, it will set a flag to the container therefore LLITE will submit it at the time when the last user drops the container. This way the OSC can recycle redundant resource and assign it to other objects.

LLITE can attach containers to file objects. When a write comes, and if the writing extent matches the extents of the container, LLITE can accomplish the I/O without creating a `cl_io`, nor does it need to take with OSC. We have done the similar thing for `fast_read`. In the initial phase, one object can only reserve one container, but it should be easily to be extended in the future.

Data Structure

```
struct cl_container_operations {
 int (*cco_hold)(struct cl_container *);
 int (*cco_release)(struct cl_container *);
 int (*cco_submit)(struct cl_container *)
};

enum cl_container_flags {
 CCF_REVOKE_PENDING = 1 << 0, /* has been revoked by OSC */
 CCF_UNCACHEABLE  = 1 << 1, /* this container can't be cached, for example
lockless IO, etc */
 CCF_REDIRTY   = 1 << 2, /* redirty pages - osc_extent already exists */
};
struct cl_container {
 atomic_t   cc_holds; /* number of users - currently 1 or 0. 1 means being
used */
 unsigned int  cc_flags;
 struct cl_object *cc_obj; /* parent object */
 int     (*cc_revoke)(struct cl_container *);
 struct cl_page_list *cc_page_list; /* list of dirty pages */

 pgoff_t    cc_index; /* first page index at file level */
 unsigned int  cc_count; /* # of pages reserved, including LRU and
{obd,osc} dirty pages */
 unsigned int  cc_grant; /* grant reserved */
 const struct cl_container_ops *cc_ops;
};

struct osc_write_container {
 struct cl_container  owc_container;
 struct list_head  owc_list;  /* list entry of container list of osc_object
*/
 struct osc_lock   *owc_lock;  /* or LDLM lock */
 struct osc_object  *owc_object;
 struct osc_io   *owc_io;  /* cl_io if it has one */
};
```

The data structure of write container should have enough information to support various operations to the container, such as submit, use/reuse, and revoke.

Create Container

Container is created for write I/O only. It must be created after the phase of `cl_io_lock()` therefore it could possibly carry a lock in the container. A new method in `cl_io_operations`, `cio_write_prepare()`, will be added to acquire a container from OSC layer.

An `cl_io` is required to acquire a container. LLITE fills in the information like the file extent it's going to write and other attributes of the I/O, and then pass those information to OSC. As usual, the LOV layer will trim the extent by the boundary of components and stripes. Once OSC receives the request, it will try to reserve the following resources for this IO:

- grant - the grant will be calculated by the chunk size
- {osc,obd}_dirty_pages
- LRU slots

It also need to take the existing cache of osc_extents into account. If the pages of writing extents are covered by existing osc_extent, this is the case of rewrite therefore we can take advantage of this by not reserving any resources, instead it will set the corresponding osc_extent to `OES_ACTIVE` and then set the container to be uncacheable({{CCF_UNCACHEABLE}}) to avoid deadlock.

The OSC may put the requesting thread in sleep when the resource is not available, and it will revoke existing containers to recycle resources. If there is no resource available and no containers existing, usually this is due to no grant available, OSC would have to create a SYNC container that a special container with 1 page in size, therefore the LLITE will have to write and issue the IO page by page.

Once the LLITE gets the container, it will fill it out by adding dirty pages into the container. A container is called full container after all resources reserved have been used. Full containers should be submitted immediately. However, LLITE can choose to cache a partial container if it decides that the container may be used later.

The OSC may not grant all credits the LLITE requested for various reasons. In that case, the LLITE layer should submit the previous full container and then request a new one.

Submit Container

A container is submitted to the OSC when it's full, or a cached container doesn't match the upcoming write extents. In the latter case, a partial container will be submitted. The resource that is not used in a container will be reused by OSC. OSC provides a callback for submitting, and there is not `cl_io` to be created to submit a container. After a container is submitted, the OSC will take it back and LLITE layer can no longer use it.

The submit callback provided by the OSC will create osc_extent if necessary and add the dirty pages into OSC I/O engine.

Revoke Container

A container can be revoked due to various reasons. Typically when a write LDLM lock is being canceled, OSC would revoke containers and write dirty pages back. OSC would also revoke containers when the resources are in shortage so that it can take reserved resources back to fulfill the requirements of new requests.

When a container is being revoked, it may be still in use by LLITE. In that case, OSC will set `CCF_REVOKE_PENDING` flag. The container will be revoked after the last holder has done using the container.

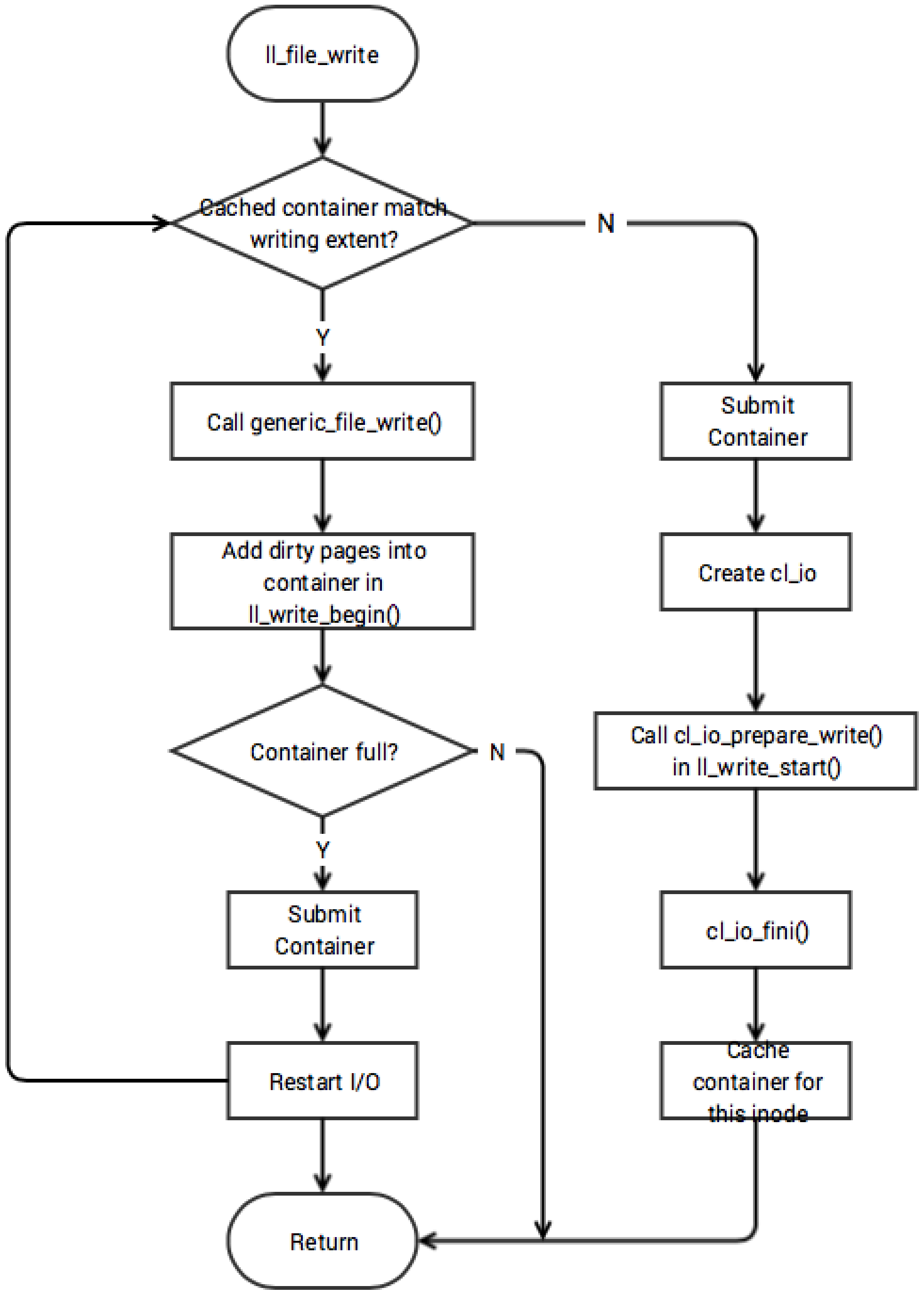
Hold/Release Container

A newly created container returned by `cl_io_prepare_write()` is in hold state; LLITE should release containers after the corresponding I/O is complete. LLITE can cache containers by keeping weak pointers to them.

Hold/Release callbacks are provided by the OSC. The major work in hold and release methods is to hold and release the corresponding LDLM lock. This is to avoid putting PtlRPC thread into sleep if the lock is canceled by OST.

Use Container in Client I/O

The client I/O stack has to be revised to use write container efficiently. In `ll_file_write()`, before a `cl_io` is created and initialized, it should check if there is any cached container for this inode. If the container includes the file extents the process is going to write, it can take advantage of this by calling kernel's generic file write directly. Then in `ll_write_begin()`, it should be able to dirty the page directly without talking to OSC.



ll_file_write
Cached container match writing extent?
N
Y
Call generic_file_write()
Add dirty pages into container in ll_write_begin()
Container full?
N
Y
Submit Container
Restart I/O
Return
Submit Container
Create cl_io
Call cl_io_prepare_write() in ll_write_start()
cl_io_fini()
Cache container for this inode

The above diagram shows the new client I/O stack after integrating with the proposal of container.

How is I/O Improved?

Small I/O performance can be drastically improved if the write extent matches with cached container, because there is no overhead of creating new `cl_io`, taking locks or adding them into OSC I/O engine. The dirty pages can be accumulated in the LLITE layer and the container will be submitted once it becomes full. Most likely this can be guaranteed if the pattern of small I/O is highly predictable.

Actually large I/O can benefit from this change as well. As what the current implement does, dirty pages can still be accumulated in the LLITE layer and submit once. However, current implementation has to add the dirty pages into I/O engine and reserve grant, etc, on a page by page basis in the OSC layer. Writer container won't have that issue, it just needs to allocate an osc_extent for newly added pages and that's it. I would expect even better performance for single thread write due to this change.

Partial Page Write

Lustre is usually defeated in bids when being asked to run workload like writing in 7KB I/O size. Lustre can't do this well because it has huge per-IO overhead and pages have to be fetched from OSTs to prepare for the partial page write. It has never been optimized for this case.

As we discussed above, the per-IO overhead can be drastically reduced by using write container. We still need to solve the problem of page read in write to make it perform well.

If the write is highly predictable, we can use read ahead to fetch pages from OSTs before partial page write occurs. Of course, this assumes that the access to the file is not in contention therefore the client can cache the write LDLM lock for relatively long time.

Some logic will be added in the VFS write interfaces like `ll_file_write()` to detect the write pattern. The simplest case is sequential write. In that case, we can invoke `ll_readahead()` to issue read ahead RPC after cl_io is initialized.

API Changes

A new method in `cl_io_operations` will be introduced:

```
struct cl_prepare_write {
 pgoff_t   cpw_start;
 unsigned long cpw_count;
 struct cl_container **cpw_cntn;
};

/**
 * Assign whatever resources are needed to complete the I/O, the container
will be
 * returned in @cntn.
 */
int (*cio_prepare_write)(const struct lu_env *env, struct cl_io_slice
*slice,
       struct cl_prepare_write *cpw);
```

And a new API `cl_io_prepare_write()` is defined to drive the above method:

```
/**
 * To reserve some resources to dirty pages.
 * OSC may not grant all the resources this IO required so this function
may be called
 * several times in the I/O.
 */
int cl_io_prepare_write(const struct lu_env *env, struct cl_io *io,
      struct cl_prepare_write *cpw);
```

The method `cl_io_commit_async()` will be retired.

`cl_page_find()` will have a new flag to tell the OSC that this page won't consume LRU slots in case this is a new page.

Operations Affected

Writepage/fsync

Obviously these functions have to flush cached containers before doing any other things, otherwise it won't be able to find the page in OSC I/O cache.

Page_mkwrite

page_mkwrite() will be revised to use new API to reserve and submit container.

LDLM Lock Cancel

When containers that are being held for use, they will take a refcount from LDLM lock, which means the lock cancel will be delayed. However, when lock cancel occurs, the OSC has to scan the list of containers and find out the overlapping containers to revoke. When an LDLM lock is being canceled, any attempt to hold the overlapping will fail.

Lockless I/O

For lockless I/O, OSC should return an uncacheable cl_container, which is a container with flag `CCF_UNCACHEABLE`, so that the container will be submitted immediately at the end of I/O.

Open Issues

N/A.