High Level Design of LAID5

2006-02-23

1 Requirements

In this work we enhance LOV layer to implement data layout pattern:LAID5. It is required to do the following:

- implement a stripe cache manager in LOV layer to cache the old stripe data and reduce the latency for updating the parity, especially the notorious pariry update via READ_MODIFY_WRITE, needing two reads and two writes in a synchronous way.
- Locking for the parity.
- IO in degraded mode.
- Inconsistent recovery.

2 Definition

First, we will introduce some terms in LAID5 (Supposed that a file is stripping over N = 3 objects via stripe pattern LAID5, as showed in the above graph) :

- stripe: Min. data unit for stripping file data, the size of a stripe is usually 1M in Lustre.
- stripe group: each stripe group contain N corresponding stripes, and one of them is parity stripe.
- stripe unit: each stripe divides up into small piece based on page-size granularity, each piece is a stripe unit.
- stripe row: stripe row is used to stand for N corresponding stripe units in a stripe group. (N - 1 data stripe units, and one parity stripe unit)

3 Functional specification

3.1 Stripe cache manager

In order to achieve the goal: calculation of parity and reconstruct data block in degraded mode, we should design a cache manager to cache the old data stripe unit and parity stripe unit on client to reduce the lantency of updating the parity. In the stripe cache manager, we need to implement the function such as add/remove/lookup operation for the stripe caches.

3.2 File Locking

To grant extent lock for IO request especially for write request, we have two choice: first, perform the extent locking operation as original , that is to say, the lock extent is same as write extent; second, all lock acquirement and cancellation are based on stripe-group-size granularity.

In the first schema, we must solve the problem of conflict for updating parity when there are multipe writers for the file which write extents are in the same stripe group. This locking strategy will be introduced in the Section 7.1.

In the second schema, we extend the lock extent to stripe-group-size granularity, that is to say, acquire a big lock covered all data strpes and corresponding parity stripe in the same stripe group. It eliminates the conflict of updating parity but reduce the concurrency.

e.g. the stripping information of a file is as follow: stripe_size: 1M, stripe_count: 4, pattern: RAID5. The stripe-group-size is stripe_size * stripe_count = 1M * 4 = 4M. If grant a extent lock [3M, 6M] in the second strategy, the extanded extent with stripe-group-size granularity is [0, 8M-1].

Two different Locking strategies result in two different strategies of updating parity which we will describe in the latter section. And in the section 7.2, we will also introduce a extended algorithm of locking stripes with stripe-group-size granularity.

3.3 IO in degraded mode

When detect just one of OSTs the objects store on is invalided, It can grace the write errors and automatically reconstruct the unaccessible block for read request.

4 Use cases

4.1 Configure the default stripe pattern

We can configure the default stripe pattern for the whole file system. Once configure the default stripe pattern of lov on MDS, MDS will use this default stripe pattern to create objects for new file.

```
#defein LOV_PATTERN_RAID5 0x04
lmc -m ....--stripepattern 4 ...
```

4.2 Set the stripe pattern via lstripe

We can also create a new file with a specific stripping pattern via lstripe.

```
usage: setstripe [-p <sttripe pattern>] <filename|dirname> <stripe  size> <stripe start
```

4.3 File IO process

- Grant extent lock with stripe-group-size granulaity.
- In the case of writeback cache:
- a. In normal case for write request, we don't update the parity immediately when dirty a file cache page correspond to a stripe row, just marking corresponding stripe unit as SU_DIRTY, and delay the parity updating until *pdflush* does batched sync, and at that time we do batched parity updating; For read request, if the read data is cached in stripe cache, we just need to copy the data unit to file cache page. And if It has already cached (N - 1) stripe units in stripe cache, we can also reconstruct the data stripe unit on client, and needn't read from OST.
- b. In the degraded case for write case, we grace the error if just one OST occures failure, just do as noraml case (marking SU_DIRTY and delay parity updating) and issue the IO to good objects; For read request, we must first read all other stripe units uncached in stripe cache form OSTs in advance and then reconstruct the read data via XORing compluation.
- In the case of Direct_IO:
- a. In the normal case for write request, we must first update the parity and then wirte the data and the parity synchronously.
- b. In the degrad case, the process is similar with writeback cache.
- After that, cancel the extent lock.

5 Logic Specifications

5.1 Stripe cache manager

Each cached stripe unit expresses as the following tuple:

```
struct stripe_unit {
    struct page *su_page;
    struct page *su_cache;
    obd_flags    su_flags;
    __u64        su_epoch;
    ...
};
```

where the *su_page* is a pointer to the corresponding file cache page, *su_cache* is old data in cache which is usually consistent with disk data.

We use *stripe_head* to stand for a stripe row which manage the stripe uints it included:

```
struct strie_head {
    struct list_head      sh_item;
    struct lov_stripe_md *sh_lsm;
    struct lov_async_page sh_pslap;
    struct page          *sh_parity;
    unsigned long         sh_index;
    atomic_t              sh_count;
    obd_flag              sh_flags;
    struct stripe_unit    sh_unit[1];
};
```

- sh_lsm: pointer to the lov_stripe_md which contains the information of file stripping.
- sh_pslap: the *lov_async_page* for parity stripe unit;
- sh_parity: the parity page, It is the redundant data, not a part of file data. the su_page and su_cache all point to it.
- sh_index: offset in the object , modulo PAGE_SIZE;
- sh_count: reference count;
- sh_unit: stripe units in the stripe row.

Similar with *address_space* managing the file cache page in linux-2.6, we can borrow *radix_tree* to manage the cached *stripe_head* of a file to implement the operation such as add/remove/lookup for the *stripe_head*.

We use data structure *ld_private_data* to manage all stripe caches of the file. The prototype is showed as follow:

```
struct ld_private_data {
     struct lov_stripe_md   *ld_lsm;
     struct obd_export      *ld_exp;
     struct radix_tree_root  ld_stripe_tree;
     spinlock_t              ld_lock;
```

```
        unsigned long           ld_nrstripes;
};
```

- ld_lsm: pointer to file striping metadata information.
- ld_exp: pointer to lov layer obd export.
- ld_stripe_tree: raidix tree of all stripe caches.
- ld_lock: spinlock to protect the radix tree. All add/remove/lookup operation for stripe_head is under the protection of this spinlock.
- ld_nrstripes: stripe count in the cache.

5.2 Lifecycle of a *stripe_head*

- Every time create a new file cache page and prepare *_async_page* for the data stripe unit, It will initialize the corresponding *stripe_unit*, and add the reference count of corresponding *stripe_head*. If the *stripe_head* is not in cache yet, we still need to create it, initilize the parity unit and prepare the *_async_page* for the parity.
- When teardown the *_async_page,* decrease the reference count of the corresponding *stripe_head*.
- Every time queue the parity *_async_page* to the update list, add the reference of corresponding *stripe_head*; In the completion handler of the parity *_async_page,* decrease the reference count.
- When the reference count become zero, teardown the *_async_page* of the parity ,and remove the *stripe_head* from the Radix_tree, release it and stripe units it included.

5.3 parity updating Method

There are two methods to update parity : READ_MODIFY_WRITE, RECONSTRUCT_WRITE.

Supposed that at the updating time one stripe row contains 4 stripe units: {s1, s2, s3, s4}, where s4 is the parity stripe unit, and s1, s3 are marked as SU_DIRTY, needing to update to the parity. si(old) stands for the old value of stripe unit i which is usually consistent with the disk data; si(new) stands for new dirtied value of stripe unit i.

5.3.1 READ_MODIFY_WRITE

The basic idea of READ_MODIFY_WRITE algorithm is as following: When the parity updating is caused by the write to one stripe unit, we first factor out the old value of the strip unit, and then calculate the new parity by XORing with the new value of the stripe unit. The following shows the processing of updating parity via method READ_MODIRY_WRITE.

- Read s1(old) and s3(old), s4 from OSTs in synchronous way if they are not in cache.
- Calculate the new parity: ('+' is short for XORing operator)

```
for an updated parity s4,
s4 = s1(old) + s2(old) + s3(old);
during updating, first factor the old value of s1:
s4 = s4 + s1(old);
calculate the new parity:
s4 = s4 + s1(old) + s1(new)
   = s1(old) + s2(old) + s3(old) + s1(old) + s1(new)
   = s1(new) + s2(old) + s3(old);
s1(update) = s1(new) + s1(old);
s3(update) = s3(new) + s3(old);
s(update) = s1(update) + s3(update) ;
s4 = s4 + s(update) =
s4 + s1(update) + s3(update)=
s4 + s1(old) + s3(old) + s1(new) + s3(new);
```

- After the calculating the parity, update the stripe cache:

```
s1(old) = s1(new);
s3(old) = s3(new);
```

The RECONSTRUCT_WRITE algorithm is:

- Read s2(old) from OST if it is not in cache;
- Update the stripe cache:

```
s1(old) = s1(new);
s2(old) = s2(new);
```

- Calculate new parity:

```
s4 = s1(old) + s2(old) + s3(old);
```

Usually we choose the updating method needing the least read operation of old data and computation of updating parity. And we borrow the module linux/md/xor.o to implement the functionality of parity calculation.

5.4 Locking callback

When flush page cache for the extent as it canceled, we also flush the parity, wait the IO finish and release the cached *stripe_heads* in the canceled extent.

5.5 Hole between objects

In the last stripe group of the file, It may be exist hole as the file size is not stripe-group-size aligned. Because recent seen size (loi->loi_rss) is usually the size of object on OBD and we get this size when grant the extent lock, So we can mark coresponding stripe_unit as SU_BLANK if sh->sh_index >> PAGE_SHIFT > loi->loi_rss. And during updating parity, we can skip the stripe unit marked as SU_BLANK.

5.6 Truncat handling

Truncate system call is used to truncate a file to a specific length. It needs special process for shrinking truncate in case of LAID5 (It needn't process extanding truncate as $0 + x = x$) . We need to remove *stripe_head* from cache and teardown corresponding *_async_page* in the truncate range. If the file size after truncate is not stripe-group-size aligned, we also need to reconstruct the parity of *stripe_head* involved truncate and update to object on OST.

6 State management

6.1 Parity updating algorithm

In this section we mainly discribe the parity updating algorithm in case of writeback cache based on the extent lock with stripe-group-size granularity.

READ_MODIFY_WRITE needs two reads (read old data and parity) before updating parity, so it will badly hurt performance while RECONSTRUCT_WRITE is very good for big write as it needn't read old data from OSTs. And our LAID implement at file level which is more complex than implementing at block device level.

The algorithm of parity updating is described as follow:

1. As mentioned above in section 4.3, when dirty the file cache page and queue asynchronous pages in function o_queue_async_io / o_queue_group_io, we just marked corresponding stripe unit with SU_DIRTY under the protection of the stripe lock(it can be the page lock of parity page *sh->sh_parity*). This flag can guide we how to update the parity when sync the file cache page. We don't queue ashnchrounous pages of parity here.

2. When add a asynchronous page of data stripe unit to the rpc list during batched syncing, we first lock the corresponding file cache page by upcall to llite layer via function .ap_make_ready, then we add this asynchronous page to the preread list of laid update group by upcall to LOV layer via function .ap_handle_stripe.

3. In the .ap_handle_stripe:

   - we first lock the stripe row (*via lock_page(sh->sh_parity)*) corresponded to this asynchronous page.

- Scan all the data stripe units, determin how to update parity according to the flags in all stripe units.
- If the syncing data stripe unit is marked as SU_UPDATE which means the old data of the stripe unit is in cache, we add the *stripe_head* to the update list of laid update group and will update parity via method READ_MODIFY_WRITE.
- If all units are marked as SU_DIRTY, it means we can update the parity via method RECONSTRUCT_WRITE. But we can not do parity updating here until all dirty file cache pages in this stripe row have synced. we just copy the data of file cache page (*su->su_page*) to the stripe cache of this data stripe unit (*su->su_cache*), clear the flag SU_DIRTY and set SU_UPDATE for the data stripe unit. And then unlock the stripe row. When syncing of last file cache page, we add the corresponding *stripe_head* to the update list of laid update group.
- Or we must preread some necessary old data and cache in the stripe manager. Mark the stripe unit needing to preread as SU_PREREAD, and add the *stripe_head* to the preread list of laid update group.

4. After finish to batch the rpc list, we do batched preread and parity updating via upcall to LOV layer via function .ap_trigger_update. In this function, we will do following operations:

- First do batched preread operation in the preread list of laid update group. After that, the cache of stripe unit (su->su_cache) is consistent with the disk, clear the flag SU_PREREAD and mark the stripe unit as SU_UPDATE.
- After preread, we shift the *stripe_heads* in preread list to update list. And then do parity updating.
- After parity updating, queue the asynchourous page of parity stripe unit in the update list, then unlock the stripe row.

5. After finish handle of parity, build rpc request for the rpc list and do the sync.

We may need to create a sepcial daemon thread to sync the parity asynchronous page to make sure the parity can sync to OST ASAP.

To reduce bad effort on performance, we'd better to updating parity via RECONSTRUCT_WRITE method. Because the smaller the stripe size is, the more IO data strips over full stripe group, so the stripe size of LAID5 should be smaller than LAID0. The best stripe size should be page size 4k; but it will add the latency for small write.

6.2 Recovery

During inconsistent recovery, failed OST need to communicate with other OSTs, so it needs to add lov/osc mouldes on OST. we can build a special recovery obd stacked on the top of lov/osc, which function is similar with llite. All process of inconsistent recovery of objects is done through this recovery obd. Because our LAID is based on object not whole OBD, so we just block the IO on the recovering object, needn't to block IO of the whole OST. In object-based recovery, we just consider two primary failure recovery: OST failure and client crash, and the algorithm is described as follow:

6.2.1 OST failure recovery

1. Similar with MDS size management in whitebook (10.9.8 MDS size management), when first write to the object on OST, write a inconsistent log record on OST contained following information: stripe pattern, fid, location information of all objects the file stripping over.

2. When last close the file in normal case, MDS cancel the inconsistent log record above.

3. If one of OSTs file stripping over occured failure (poweroff), reboot and rejoin to the cluster, It first scans the inconsistent log record, execute open operation on the suspicious objects existed inconsistent problem and mark their inodes as the flag OBD_INCONSIST. And then process the generic recovery with MDS and client such as replaying uncommited request and lock server etc.

4. After the normal recovery, recovery obd begin to connect to all other OSTs and start inconsistent recovery .

5. During recovery, All IO requsts on objects marked as OBD_INCONSIST return failure immediately, read retry on client can grace the error. And all operations on other objects are as normal.

6. When do recovery for an inconsistent object, recovery obd first grants extent lock PR[0, -1] on all other corresponding objects the inconsistent file stripping over. Then read data from these objects and reconstruct the data of the inconsistent object.

7. If one of OSTs other objects store on occurs failure (such as power off, diconnect with the recoverying OST or connection failed when recovery obd setups) during recovery (we call it double failure), stop recovery for the inconsistent object immediately. After the double failed OST rejoin to cluster, do the inconsistent recovery for the object again.

8. After inconsistent recovery, cancel the extent lock and clear the flag OBD_INCONSIST for the object . If the file is not in write context, delete the inconsistent log record.

9. If multiple OSTs occured failure nearly sychronously, there may be multiple objects of the file marked as OBD_INCONSIST. At this time, the objects of the file marked as OBD_INCONSIST will involve in a election process. we always choose object with earlier modified time to do the data reconstruction for inconsistent recovery.

6.2.2 Client crash recovery

The crash of the client may cause inconsistent write, too. In this case, we can do inconsistent recovery according to the extent lock in OST's lock namespace. When detect the client crash, It will cancel all extent lock grant by the crash client in the eviction handler on OST. If stripe pattern of the file granted the extent lock is redundant pattern such as LAID1, LAID01 and LAID5, we will do the inconsistent recovery by the recovery obd or the dedicated recovery client.

The recovery of OST failure can be also based on the log record of the extent lock.

7 Alternative

7.1 Update parity on OST

In the above LAID5 design, the extent lock is stripe-group-size granularity. We must always grant lock form lock servers of all OSTs the file stripping over. If one of OSTs goes down, the lock acquirement will block and may result in the eviction of clinet. So extent lock with stripe_group-size granularity may be not a good idea; but It elimates the conflict for updating parity, and is good for IO in the degraded mode as we must grant lock with stripe-group-size granulairty or covered the stripe row at least to reconstruct unaccessable stripe unit data.

We propose an another strategy: updating parity on OST.

It's based on the following idea: A + B + C = A + C + B, that is to say executing XOR operation out of order can get the same result.

The algorithm is as follow:

- Client acquires lock with extent same as write extent, don't acquire any lock for parity.
- On client when do batched parity updating, If the data units in the stripe row are all in cache (we have already grant extent lock covered the stripe row), update the parity via RECONSTRUCT_WRITE, and mark the parity as PARITY_OVERWRITE; If the stripe units are not all in cache, we calculate the D(update) = D(old) + D(new), mark it as PARITY_UPDATE, and send the D(update) to OST, and leave the left updating work to OST.
- On OST If the received parity data is marked as PARITY_OVERWRITE , just write it to OBD; If the parity data is marked as PARITY_UPDATE

(P(update)), we should read the old pairy (P(old)) and calculate the new parity P(new) = P(old) + P(update), and then write it to OBD.

- For read in degraded mode, to reconstruct the data on failed OST we still need to grant extent lock covered the stripe row or with stripe-group-size.

To reduce the OST's workload and lantency of updating parity , we would better to cache the parity pages on OST.

7.2 Master Locking for the file

In the previous design of mirror LAID, client acquires the lock from lock servers of OSSs the mirrors store on; In the above design of LAID5, we must acquire lock with stripe-group-size granularity from lock servers of OSSs all objects store on. Now we introduce a new strategy : Choose an OSS one of objects stores on as a master lock server.

For LAID1, we just need to acquire the extent lock from the master lock server; For LAID5, we just need to acquire the lock with stripe-group-size granularity on master lock server, then we think we have acquire the lock with same extent from all other servers the stripping objects store on. Via this way, It can reduce the lantency of lock acquirement but involve the remastering of lock server when it occurs failure.

The remastering algorithm in case of LAID1/LAID5 is described as follow:

- When the client opens a file for r/w, the MDS will return the index of master object (usually the first object). After that the client just takes the extent lock from the server the master object stores on with stripe-group-size granularity.
- When one request of lock acquisition for a file IO occurs time-out and detect the master lock server occurs failure, the client will first flush the file cache, cancel the extent lock took from master lock server and start the remastering process.
- The client send message REMASTER_LOCK_SRV to notify the MDS to coordinate the remastering process of the file.
- The MDS first choose another object as a master object, and send message COORDINATE_MASTER_START with the choosen master object index to clients opened this file. (At this place we may borrow the way that add OST dynamically via the callback locking to notify the clients open this file to remaster the master object).
- when client receive the message COORDINATE_MASTER_START, it first flush the file cache, cancel the extent lock grant from the failed master lock server, replace the master object with the one the message passes in, block every lock request for the file. After that, send message CLI_REMASTER_FINI to MDS.

- when MDS receive all reply messages from the clients, send message MDS_REMASTER_FINI to all client involved the remaster process.
- When client receives the message MDS_REMASTER_FINI, it cancels the blocking. After that, all operation is as normal.

8 Focus for inspections

- Is the design reansonable ?
- Is there any leak? especially on support to mmap write via new algrothim of parity updating.
- Which strategy is better, taking lock with stripe-group-size or udating parity involved OST? Is there any other better way to implement LAID5?