High Level Design for Aggregate Stat

Fan Yong

2010-10-10

0.1 Introduction

When using Lustre under high latency network environment (like WAN), the performance of traversing large directory (like "ls -l", hereinafter the "traversing thread") is so bad as to be almost intolerable. There are test results from customer:

```
Time for "ls -l" a directory with 100K files (single stripe)
Local: 16.7 seconds
Remote with 2.5 ms latency: 286 seconds
Remote with 35 ms latency: 3528 seconds
```

Consider the remote operation, the traversing time is approximately equal to "RPC_latency * item_count". We can conclude that most of the time is wasted on such RPC latency. There are several possible solutions for improving such traversing performance:

1. Decrease network latency

   If we could control such network latency, such as try different router(s), it maybe much helpful. Unfortunately, for most cases, especially under WAN environment, the network latency is affected by many facets, and it is almost impossible to be controlled by end user.

2. Metadata read-ahead and fully parallel process

   Lustre uses statahead for metadata read-ahead, if without such feature, the above performance will be 40% ~ 50% worse. In Lustre, file attributes are divided into two parts, one part (including mode, owner, stripe information, ACL and etc) is on MDS, the other part (including size, xtime and etc) is on OSS. Statahead just pre-fetches file attribute from MDS, the traversing thread fetches the other file attribute from OSS by itself when needed. We assume that the RPC time for "client<=>MDS" and "client<=>OSS" are the same, so fully parallel statahead can save at most 50% of the whole traversing time (comparing with non-statahead case).

   More explanation about the fully parallel statahead: the statahead thread and the traversing thread deal with their business in parallel, and can be described as the model of producer and consumer. If the statahead thread (producer) can not per-fetch file attribute from MDS quickly enough, then the traversing thread (consumer) has to wait for the statahead thread. In Lustre, there are limitation for in flight RPC count between client and MDS. If the statahead thread can pre-fetch file attribute quickly enough, such limitation will not much affect in flight statahead RPC, and the statahead thread can feed the traversing thread quickly enough. But under high latency network environment, during one RPC latency, the statahead thread maybe send out more statahead RPCs than such limitation, then some statahead RPCs have to queue until former statahead RPCs replied. That is not fully parallel. So under such case, we should increase such

in flight RPC count limitation, to allow more in flight (statahead) RPCs between client and MDS. On the other hand, large count in flight RPC maybe cause MDS overload, so there should be some balance for such tuning.

As you have seen above, for one stat operation from userspace, it triggers at least two RPCs (ignore cached case), the first one is for file attribute from MDS (including stripe information), the second one is for file size/xtime from OSS (according to stripe information). Currently, the statahead thread only processes the first RPC, if it can process the second one also, means pre-fetches file attribute from OSS, then the performance of traversing large directory will be much improved.

3. Size on MDS (SOM)

   If allow MDS to cache file size/xtime attribute, then client only needs one RPC to fetch all the file attribute from MDS. Consider the best cases, such cache is valid for all files under a directory, then traversing such directory can save near 50% time. SOM is just for that, such feature is in processing for years by Oracle. Unfortunately, it is some complex as to there is no stable version can be used yet.

4. Reduce RPC count

   Currently, one lookup/getattr RPC is just for attribute of one file, if we can combine several lookup/getattr RPCs into one, then client can fetch multiple files attribute with single RPC. The similar situation for glimpse RPC between client and OSS. Assume the average RPC aggregate degree is "N" (count of items/files are processed in one RPC), then only need 1/N RPCs for traversing the same directory. If high network latency is the unique main reason for bad performance of traversing large directory, then it is significant improvement. That is the basic idea for aggregate stat. It will run together with statahead.

0.2 Requirements

- The performance of traversing large directory (contains 100K single stripe files) on high latency (> 1 ms) network should be improved at least 100%.
- The performance of traversing large directory (contains 100K single stripe files) on local network should not be worse.
- The performance of traversing small directory (not more than 1K files) should not be worse.

All the tests should be done on single client, and no other writers on other client(s) under the same directory.

0.3 Functional specification

1. User control interface for enable/disable statahead

   There is already such proc interface on client side, name "statahead_max". "0" means disable, "N" means the max statahead window size.

2. User control interface for enable/disable statahead from OSS

   It is a new proc interface on client side, named "statahead_oss". Because glimpse RPC maybe not obtain related extent lock(s) from OST(s) if some other(s) is (are) changing related OST object(s). Under such case, the size obtained by statahead is transient one, the traversing thread has to re-fetch file size by itself when really used, means the glimpse RPC triggered by the statahead thread is redundant. Such interface allows the user to enable/disable statahead from OSS according to the real cases. "0" means disable, others means enable, "statahead_oss" is by pass if "statahead_max" is "0".

3. User control interface for enable/disable aggregate stat

   It is a new proc interface on client side, named "statahead_aggregate". "0" means disable, "N" means the max value RPC aggregate degree, "statahead_aggregate" is by pass if "statahead_max" is "0". Will discuss this interface more detailed in logic specification.

4. Other statistics interfaces for aggregate stat

   Including how many aggregate stat RPCs triggered, average aggregate degree, and so on.

0.4 Use cases

There are several facets should be considered for testing and comparing, those should be combined.

- directory size

  1K items, 10K items, 100K items

- stripe count

  full stripe, single stripe, zero stripe (sub-dir, or file with "O_LOV_DELAY_CREATE"), misc cases

- network latency

  local network, 1 ms latency, 10ms latency

- enable/disable statahead/aggregate stat

  disable statahead, enable statahead from MDS only without aggregate stat, enable statahead from OSS without aggregate stat, enable statahead from OSS and aggregate stat

0.5 Logic specification

0.5.1 Aggregate degree

Means how many items can be processed within one aggregate stat RPC. It is not more large value more good. There are five main facets about the degree of aggregate getattr RPC between client and MDS:

- page size

  We only want to aggregate the items within one readdir() page from MDS, because more items across multiple readdir() pages maybe cause readpage RPC(s), such latency can not be ignored under high latency network environment. Consider readdir() page from MDS, each page contains several entries:

```
struct ll_dir_entry {
        /* number of inode, referenced by this entry */
        __le32  lde_inode;
        /* total record length, multiple of LL_DIR_PAD */
        __le16  lde_rec_len;
        /* length of name */
        __u8    lde_name_len;
        /* file type: regular, directory, device, etc. */
        __u8    lde_file_type;
        /* name. NOT NUL-terminated */
        char    lde_name[LL_DIR_NAME_LEN];
};
```

  Assume the average item name length is “L”, then average entry size in readdir() page is “L+8”, so the entry/item count in one readdir() page is “C”=“PAGESIZE/(L+8)”. The max aggregate degree “N” must be less than “C”. And if “L” is 2 (the min value), PAGESIZE is 4096 (normal case), then “N” is less than 410.

- RPC size and reply size

  Currently, the max RPC size (limitation in LNET layer) is not more than 1M bytes. For an item with full stripe and full ACL set, the reply size is about 4.5K (ldlm_reply + mdt_body + EA + ACL), so under such case, “N” is not more than 230.

- MDS load and client impartial

  Aggregate getattr is large RPC, one aggregate getattr RPC equals to “N” normal lookup/getattr RPC. From the view of MDS, it causes related service thread to hold CPU for longer time than other normal RPC, and then causes other RPC to queue for more time. Even serving aggregate getattr with separated portal on MDS, it also maybe cause other aggregate getattr RPCs (from the same or different client(s)) starved.

- LRU size for cached ldlm lock on client side

  For aggregate getattr RPC reply, MDS returns "N" IBITS locks if all of them are available at once without waiting. And then, client must caches such locks on client side. So more large "N", means more IBITS locks cached on client side, once reach the LRU size limit, then some of them will be dropped by client, even though before they are used by the traversing thread. So the traversing thread has to send lookup/getattr RPCs again by itself when really access these items.

- hit rate and statahead window size

  Statahead is based on prediction. In fact, the statahead thread does not know whether the caller want to traverse the whole directory or not. So it just pre-fetches some items, but not all, and goes ahead step by step. That is the statahead window. If the subsequent stat operations hit the items which are pre-fetched, then can enlarge statahead window, otherwise maybe shrink statahead window or stop statahead (depends on the miss rate). The aggregate getattr should be controlled by such statahead window also, mean the degree of aggregate getattr should not larger than statahead window size.

So we should set a max "N" by force (Currently, I prefer to "N<=48"), within such limitation, users on client can adjust "N" by proc interface. To simplify the design and implementation, aggregate glimpse share the same aggregate degree "N" with aggregate getattr. But it is some different for aggregate glimpse RPC, a bit complex. Because the OST count is more than MDT count (even if consider CMD case in future) for general cases. If most items are single striped in the directory and OST count is some large, then it is not easy to wait every OSC's aggregate degree reach "N" and then trigger the aggregate glimpse RPC, otherwise the glimpse requests which are added into such aggregation early have to wait for some long time, even exceed RPC latency. So we uses more flexible policies to trigger aggregate stat RPC, which will be discussed in later section.

0.5.2 Aggregate policies

Aggregate RPC means some requests wait in the aggregation until it is full or some other condition(s) is/are triggered. But it should not cause the early joined requests to wait for so long time as to exceed RPC latency. So we should set some conditions to trigger aggregate stat RPC.

- general policies
  - Once sub-request count reaches the defined aggregate degree, then triggers aggregate stat RPC.
  - Before the statahead thread triggers any other RPC (except aggregate RPC), if the aggregation is not empty, then triggers aggregate stat RPC.

  - Before the statahead thread tries to obtain any local lock (like semaphore), which will cause the statahead thread to wait without CPU held, if the aggregation is not empty, then triggers aggregate stat RPC.
  - When comes to the end of the directory, if the aggregation is not empty, then triggers aggregate stat RPC.
  - If the statahead thread exit abnormally (killed by others, or high miss rate, or something wrong), then drops the aggregation.

- special policies for aggregate getattr

  - Only the lookup/getattr request triggered by the same statahead thread can be added into the same aggregation.

    One reason is that these lookup/getattr requests can share the same parent directory information, sponsor information, which need to be sent to MDS for lookup/getattr related child items. So reduces RPC size. The other reason is that the statahead thread should process the aggregate getattr RPC reply, to build/update related dentry/inode in cache, to trigger related glimpse request. Based on current statahead mechanism (one statahead thread only serves one traversing thread), lookup/getattr request triggered by different statahead threads should be processed separately. So for each MDC, there maybe several aggregate getattr RPCs to be triggered at the same time.

- special policies for aggregate glimpse

  - All the glimpse requests triggered by any statahead thread can be added into the same aggregation.

    Each glimpse request is different from others, there is no common information can be shared among them, even though for these objects in the same parent directory. On the other hand, the aggregate glimpse RPC reply can be processed by the registered callback functions without any statahead thread involved. Another advantage is that it can accelerate the gathering of asynchronous glimpse requests, and reach to the defined aggregate degree more quickly, and then to be triggered. So for each OSC, there is only one aggregate glimpse RPC to be triggered at the same time.
  - Any glimpse request triggered by other non-statahead thread can join the aggregation under related OSC, and triggers the aggregate glimpse RPC at once.

    Make full use of any can be shared RPC to deliver the aggregation. There is at most one sub-request in the aggregate GLIMPSE RPC which is triggered by non-statahead thread.

0.5.3 Protocol changes

1. Introduce new RPC named “MDS_AGGREGATE_GETATTR” for aggregate getattr between client and MDS

- RPC request format

  header (lustre_msg_v2)|ptlrpc_body|mdt_body|sub_req_desc|sub_req(s)

  – header: the same as other RPC, all the sub requests are counted as one field
  – ptlrpc_body: "pb_opc" is MDS_AGGREGATE_GETATTR
  – mdt_body: "fid1" is parent fid, "fid2" is unused
  – sub_req_desc: sub-request count, and array for each sub-request length

```
struct sub_req_desc {
    __u32 srd_count;
    __u32 srd_lens[0];
}
```

  – sub_req: it is simplified ENQUEUE (LOOKUP/GETATTR) RPC request

```
sub_req_header|ldlm_request|child_fid (or filename)
struct sub_req_header {
    __u32 srh_count;
    __u32 srh_opc;
    __u32 srh_valid;
    __u32 srh_lens[0];
}
```

- RPC reply format

  header (lustre_msg_v2)|ptlrpc_body|sub_rep_desc|sub_rep(s)

  – header: the same as other RPC, all the sub replies are counted as one field
  – sub_rep_desc: sub-reply count, and array for each sub-reply length

```
struct sub_rep_desc {
    __u32 srd_count;
    __u32 srd_lens[0];
}
```

  – sub_rep: it is simplified ENQUEUE (LOOKUP/GETATTR) RPC reply

```
sub_rep_header|ldlm_reply|mdt_body[|EA|ACL]
struct sub_rep_header {
    __u32 srh_count;
    __u32 srh_valid;
    __u32 srh_lens[0];
}
```

- RPC portal

  - request portal: MDS_AGGREGATE_PORTAL
  - reply portal: MDS_AGGREGATE_PORTAL

  Do not share other service portals on MDS. Because "MDS_AGGREGATE_GETATTR" is large RPC, which needs more large request/reply buffer than other RPC. On the other hand, it maybe hold MDS CPU for more long time as to cause other RPC starved under share portal mode.

2. Introduce new RPC named "OST_AGGREGATE_GLIMPSE" for aggregate glimpse between client and OST

   - RPC request format

     header (lustre_msg_v2)|ptlrpc_body|sub_req_desc|sub_req(s)

     - header: the same as other RPC, all the sub requests are counted as one field
     - ptlrpc_body: "pb_opc" is OST_AGGREGATE_GLIMPSE
     - sub_req_desc: sub-request count, and array for each sub-request length

       ```
       struct sub_req_desc {
           __u32 srd_count;
           __u32 srd_lens[0];
       }
       ```

     - sub_req: it is simplified glimpse RPC request

       ```
       sub_req_header|ldlm_request
       struct sub_req_header {
           __u32 srh_count;
           __u32 srh_opc;
           __u32 srh_valid;
           __u32 srh_lens[0];
       }
       ```

   - RPC reply format

     header (lustre_msg_v2)|ptlrpc_body|sub_rep_desc|sub_rep(s)

     - header: the same as other RPC, all the sub replies are counted as one field
     - sub_rep_desc: sub-reply count, and array for each sub-reply length

       ```
       struct sub_rep_desc {
           __u32 srd_count;
           __u32 srd_lens[0];
       }
       ```

     - sub_rep: it is simplified glimpse RPC reply

```
sub_rep_header|ldlm_reply|ost_lvb
struct sub_rep_header {
    __u32 srh_count;
    __u32 srh_valid;
    __u32 srh_lens[0];
}
```

- RPC portal
  - request portal: OST_AGGREGATE_PORTAL
  - reply portal: OST_AGGREGATE_PORTAL

3. Introduce new ldlm lock flags "LDLM_FL_STATAHEAD" to indicate whether the lock request is triggered by statahead thread

0.5.4 MDS side logic

When MDS receives aggregate getattr RPC, it analyzes and processes every sub-request in turn.

- For the first sub-request in the aggregation, MDS can process it as normal lookup/getattr RPC (with intent lock) without reply at once. If someone has held related IBITS lock for the first item, and caused the service thread blocked, then just wait until obtained such lock.
- For other sub-requests, process them as normal lookup/getattr RPC (with intent lock) without reply at once, but if someone has held related IBITS lock for some item, and will cause the service thread blocked, then just stop processing (LDLM_FL_BLOCK_NOWAIT), ignore all subsequent sub-requests in the aggregate getattr RPC. Because we have held at least one IBITS lock already, if blocked and wait for other IBITS locks, it maybe cause deadlock. Under such case, just reply the client with the successful processed item(s). Client will send new RPC(s) later for those unprocessed items yet.

0.5.5 OSS side logic

When OST receives aggregate glimpse RPC, it analyzes and processes every sub-request in turn. There are two cases:

- For those sub-requests without conflict lock(s) held by other client(s), just process them as normal glimpse RPC without reply at once.
- For those sub-requests with conflict lock(s) held by other client(s), if it is triggered by non-statahead thread, then processes it as normal glimpse RPC (sending glimpse callback to the highest lock holder to query file size if needed), otherwise ignores such sub-requests (just uses the stale size on such OST, because

the traversing thread will fetch file size by itself later if without glimpse lock granted here). Since there is at most one sub-request in the aggregate glimpse RPC which is triggered by non-statahead thread, the whole processing of the aggregate glimpse RPC will trigger glimpse callback at most once, other operations are all locally as normal glimpse RPC does. So it will not introduce any additional latency by network.

After all the sub-requests have been processed, packs all the results and replies.

0.5.6 Client side logic

There are three main entities on client side for aggregate stat operation: the traversing thread, the statahead thread and the glimpse callback.

- the traversing thread

  When the traversing thread begins to traverse the directory, it triggers the statahead thread firstly. Then it stats every item in the directory in turn. During those stat operations, it maybe wait because of related item(s) unready until the statahead thread or glimpse callback wakes it up.

  The traversing thread controls the statahead window. If the item it wants is ready in cache without further lookup/getattr RPC to MDS, then it hits, and then the statahead window maybe enlarged, otherwise for miss cases, it maybe shrink the statahead window or just stops the statahead thread. Whether hit or miss, after each stat operation, the traversing thread will move forward the statahead window and try to wake up the statahead thread if it is waiting on the statahead window boundary.

- the statahead thread

  1. The statahead thread fetches directory page(s) from MDS and scans such page(s). For each item, it make a lookup/getattr request down to the lower layer MDC (dispatched through LMV for Lustre-2.x).
  2. In MDC layer, checks whether held related IBITS lock already, if yes, moves such item to glimpse process, otherwise adds such request into related aggregation (identified with statahead ID), if such aggregation is full, triggers aggregate getattr RPC.
  3. When reaches statahead window boundary, or end of the directory page (need more page from MDS), the statahead thread will trigger aggregate getattr RPC if related aggregation is not empty, and then process former aggregate getattr RPC reply. For those items without related IBITS locks obtained, re-adds them into related aggregation. For items gained related IBITS locks from MDS, makes a glimpse request down to the lower layer OSC (dispatched through LOV) for each.

4. In OSC layer, checks whether held related EXTEND lock already, if yes, callback and wakes up the traversing thread (if all other related EXTEND locks held), otherwise adds such request into the OSC aggregation, if the aggregation is full, trigger aggregate glimpse RPC.
5. After processing all the aggregate getattr RPC replies and triggering all related aggregate stat RPCs (whether reaches aggregate degree or not), the statahead will trigger readpage RPC for end of directory page case, or fall into waiting if reaches statahead window boundary until be waken up by the traversing thread.
6. Repeat above process until the statahead thread is stopped. For normal exit (means reaches the end of directory), all the items must be processed (aggregate glimpse RPC triggered if needed). For other abnormal cases, drops related aggregation.

- the glimpse callback

  It processes aggregate glimpse RPC reply, updates inode size/xtime, and wakes up the traversing thread. Because glimpse operation is non-blocked, it does not guarantee to obtain related EXTENT lock together with object size/xtime information. It is the traversing thread's duty to check whether held related EXTENT lock or not after waken up, if without, sends glimpse RPC again by itself.

0.5.7 Resend aggregate stat RPC

If something wrong caused the aggregate stat RPC reply lost, then related aggregate stat RPC will be resent and reprocessed.

- client

  Just resend related aggregate stat RPC, no special process.

- OSS

  Because glimpse RPC is non-blocked operation, then OST maybe not grant lock to the client if related lock is in using by others. When OST processes the resent aggregate glimpse RPC, it needs to check compatible queue again to make sure whether the conflict lock still is in using or not.

- MDS

  Because MDS maybe only processed part of the sub-requests in the original aggregate getattr RPC (for LDLM_FL_BLOCK_NOWAIT or others), then for those have been processed, reprocesses them as normal resent lookup/getattr RPC (with intent lock) without reply at once; and for others, processes them as new lookup/getattr RPC (without reply at once).

0.5.8 Replay aggregate stat RPC

According to current Lustre recovery mechanism, LOOKUP/GETATTR/GLIMPSE RPC are all idempotent operations, in this sense, no need to replay aggregate stat RPC. On the other hand, client maybe hold some ldlm lock(s) through aggregate RPC, these locks can be replayed as normal ldlm lock recovery process if needed, nothing special related with aggregate stat RPC.

0.5.9 Scalability & performance

Aggregate stat is used for improving the performance of traversing large directory on single client under high latency network environment. But from the view of server, aggregate stat RPC is large one, it increases the server load. If many clients trigger lots of aggregate stat RPCs at the same time, then maybe cause server overload. Under such case, from the view of client, the performance of traversing large directory maybe not as good as expected, even worse. So some new mechanism should be introduced to allow server to tell client that the server is in high load mode, do not send large aggregate stat RPC, means decreasing the aggregate degree. A simple way for that is to return some flags in the aggregate stat RPC reply.

Further study the issues under large scaled Lustre system, it is about how to establish the aggregate degree. It is not only the client decision, but also the negotiation result with server, and can be adjusted dynamically according to the system load.

It is obvious that the statahead window size should not be smaller than the aggregate degree. To make the statahead thread fully run with less waiting for server, the statahead window size is at least double the aggregate degree. Then when one aggregate stat RPC is in processing by server, client can fill another aggregate stat RPC at the same time.