High Level Design for Aggregate Stat

Fan Yong

2010-12-29

0.1 Introduction

In lustre, sometimes, the performance of traversing large directory (like "ls -l", hereinafter the "traversing thread") is so bad as to be almost intolerable, especially under high latency network environment (like WAN). There are some test results:

> *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. 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 stored on the OSTs. Statahead just pre-fetches file attributes from MDS. The traversing thread fetches the other file attributes from OSTs by itself when needed. We assume that the RPC time for "client<=>MDS" and "client<=>OSTs" 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) cannot per-fetch file attributes from MDS quickly enough, then the traversing thread (consumer) has to wait for the statahead thread. In Lustre, there is limitation for in flight RPC count between client and MDS. If the statahead thread can pre-fetch file attributes 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 balances 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 attributes from MDS (including stripe information), the second one is for file size/xtime from OSTs (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 attributes from OSTs, then the performance of traversing large directory will be much improved.

2. Size on MDS (SOM)

   If allow MDS to cache file size/xtime attributes, then client only needs one RPC to fetch all the file attributes from MDS. Consider the best case, such caches are valid for all files under the 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 no stable version can be used yet.

3. Aggregation

   Currently, one lookup/getattr RPC is just for attributes of one file, if we can combine several lookup/getattr RPCs into one, then client can fetch multiple files attributes with single RPC. The similar situation for glimpse size RPC between client and OSTs. 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. The other advantage of aggregation is that it can give better performance for traversing large directory if servers are under heavy load and the stat RPCs have to queue behind many other RPCs in the network.

4. Multiple directory pages in one readdir RPC

   Currently, for each readdir RPC, client only fetches one directory page, which is inefficient. If allow client to read multiple directory pages in one readdir RPC, means reducing readdir RPC count compared with normal case, then the performance of traversing large directory (especially for simple list items name in the directory without other attributes under high latency network environment) will be improved.

5. Readdir+

   For normal readdir RPC, MDS only returns directory content (items with their names, identifiers, and offset/hash). As for other attributes of directory items, like owner, mode, stripe information, and etc, the traversing thread will fetch them (with related ldlm lock) from MDS with separated LOOKUP/GETATTR RPC.

   For readdir+ RPC, MDS will return all the attributes stored on MDS for the directory items to client (with/without related ldlm locks granted). Consider the best case, if no conflict modifications under the same directory when traverses such directory, no additional separated LOOKUP/GETATTR RPC as normal traversing directory does. For large directory, the items that are under modifying when readdir+ occurs are relative seldom compared with the items count in the same directory, so readdir+ can reduce almost all the LOOKUP/GETATTR RPCs for traversing large directory, then related performance will be much improved.

On the other hand, because more attributes are returned by readdir+ RPC, the original readpage mode (one page one RPC) is not suitable, we should allow client to read multiple directory pages in one readdir+ RPC, just as described above.

As described above, 3/4/5 all can reduce RPC count for traversing directory. In fact, 4 is relative easy to be done, Lsy will make basic implementation for that. In this design document, we mainly focus on aggregation. As for readdir+, there will be separated HLD for it.

0.2 Requirements

Aggregate stat includes two parts: aggregate getattr and aggregate glimpse size. If both features are enabled, the performance of traversing large directory (contains 1M single-striped items) under high latency (> 1 ms) network environment should be 200% improved at least. And more latency more improvement. Even if on local network, there should be about 100% improvement.

Note: there should not other modification operations on other client(s) under the same directory when traverses.

0.3 Functional specification

1. Administrator 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. Administrator control interface for enable/disable statahead for OSS

   It is a new proc interface on client side, named “statahead_oss”. Because glimpse size 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 uses, means the glimpse size RPC triggered by the statahead thread is redundant. Such interface allows the administrator 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. Administrator 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 specifications.

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

  100K items, 1M items and 10M items

- stripe count

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

- network latency

  local network, 1 ms latency and 10ms latency

- enable/disable statahead/aggregate stat

  only enable statahead without aggregate, only enable aggregate for MDS, enable aggregate for both MDS and OSS

0.5 Logic specifications

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:

- 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 that 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.

- 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 long 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.

So we should set a max "N" by force (Currently, I prefer to "N<=64"), within such limitation, users on client can adjust "N" by proc interface. To simplify the design and implementation, aggregate glimpse size shares the same aggregate degree "N" with aggregate getattr. But it is some different for aggregate glimpse size 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 to reach "N" and then trigger the aggregate glimpse size RPC, otherwise the glimpse size requests which are added into such aggregation early have to wait for some long time, even exceed RPC latency. So we use more flexible policies to trigger aggregate stat RPC, which will be discussed in later section.

0.5.2 Aggregate policies

Aggregate stat 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 the information to be transferred. 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 size 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 size

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

    Each glimpse size 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 size 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 size 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 size RPC to be triggered at the same time.
  - Any glimpse size request triggered by other non-statahead thread can join the aggregation under related OSC, and triggers the aggregate glimpse size 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 size RPC that 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.

   Both the aggregate getattr sub-requests and sub-replies are transferred as bulk pages. Because filename length is flexible, normal RPC request buffer only can contain very limited items with long filenames, it is not worth to prepare large buffer for such uncertain situations, bulk pages are more suitable. The similar case is for file attributes with EA and ACL, which sizes are flexible also.

   - RPC request format

     header (lustre_msg_v2)|ptlrpc_body|mdt_body|sub_req_desc

     - header: the same as other RPC
     - ptlrpc_body: "pb_opc" is MDS_AGGREGATE_GETATTR
     - mdt_body: "fid1" is parent fid, "fid2" is unused, "size" is reused as sub_request count, "nlink" is reused as client side prepared buffer size for reply, "generation" is reused as client side buffer size for request.
     - sub_req_desc: a variable length array to describe each sub-request length

     The sub-requests are simplified ENQUEUE (LOOKUP/GETATTR) RPC requests, and transferred as bulk pages. They are packed continuously (64bits aligned) in these pages, and allow them to cross pages. The format for each sub-request is as following:

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

     Different items maybe have different attributes sizes, client only can estimate the max size for each item. "mdt_body->nlink" is for the total buffer size. Data for aggregate getattr reply should not larger than such size. It is no problem if those buffers are not full.

   - RPC reply format

     header (lustre_msg_v2)|ptlrpc_body|sub_rep_desc

     - header: the same as other RPC
     - ptlrpc_body: the same as other RPC
     - sub_rep_desc: a variable length array to describe each sub-reply length

The sub-replies are simplified ENQUEUE (LOOKUP/GETATTR) RPC replies, and transferred as bulk pages. They are packed continuously (64bits aligned) in these pages, and allow them to cross pages. The format for each sub-reply is as following:

sub_rep_header|ldlm_reply|mdt_body|[mdt_md]|[ACL]
struct sub_rep_header {
__u32 srh_count;
__u32 srh_lens[0];
}

- RPC portal
  The same as normal readdir RPC.
  - request portal: MDS_READPAGE_PORTAL
  - request bulk portal: MDS_BULK_PORTAL
  - reply portal: MDS_READPAGE_PORTAL
  - reply bulk portal: MDS_BULK_PORTAL

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

   It is unnecessary to transfer aggregate glimpse size sub-requests and sub-replies with bulk pages, which is different from aggregate getattr. Because for each sub-request and sub-reply, the format and size are fixed, the caller can calculate the needed buffer exactly. And the size for each sub-request (ldlm_request: 104 bytes) and sub-reply (ldlm_reply: 112 + ost_lvb: 40 bytes) is relative small, based on currently RPC pre-allocated buffer mechanism (OST_MAXREQSIZE: 5 * 1024 bytes, OST_MAXREPSIZE: 9 * 1024 bytes), single aggregate glimpse size RPC can contain more than 40 items. For most cases, it is enough. Maybe you want to set more large aggregate degree, but as described above, there are more chances to trigger aggregate glimpse size RPC than aggregate getattr RPC. For many cases, the RPC was triggered before the aggregation is full. Under such cases, it is more efficient than using bulk pages.

- RPC request format

  header (lustre_msg_v2)|ptlrpc_body|sub_req(s)

  - header: the same as other RPC, each sub-request is counted as one field
  - ptlrpc_body: "pb_opc" is OST_AGGREGATE_GLIMPSE, "pb_pre_versions[0]" is reused as sub-request count
  - sub_req: it is simplified glimpse size RPC request

    sub_req_header|ldlm_request
    struct sub_req_header {
    __u32 srh_count;
    __u32 srh_flags;
    __u32 srh_lens[0];
    }

- RPC reply format
  header (lustre_msg_v2)|ptlrpc_body|sub_rep(s)
  - header: the same as other RPC, all the sub-replies are counted as one field
  - ptlrpc_body: "pb_pre_versions[0]" is reused as sub-reply count
  - sub_rep: it is simplified glimpse size RPC reply

    sub_rep_header|ldlm_reply|ost_lvb
    struct sub_rep_header {
    __u32 srh_count;
    __u32 srh_lens[0];
    }
- RPC portal
  - request portal: OST_REQUEST_PORTAL
  - reply portal: OST_REQUEST_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 size 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 size 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 size RPC (sending glimpse callback to the highest lock holder to query file size if needed), otherwise ignores such sub-requests (just uses the staled 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 size RPC that is triggered by non-statahead thread, the whole processing of the aggregate glimpse size RPC will trigger glimpse callback at most once, other operations are all locally as normal glimpse size 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 size 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 size 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 size 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 size RPC triggered if needed). For other abnormal cases, drops related aggregation.

- The glimpse callback

  It processes aggregate glimpse size RPC reply, updates inode size/xtime, and wakes up the traversing thread. Because glimpse size 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 size 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.

- OST

  Because glimpse size 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 size 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 stat 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, especially under high latency network environment. But if there are many modification operations under the same directory when traverses such directory, then pre-obtained ldlm IBITS lock(s) maybe canceled by MDS before really used. Means related RPC(s) for pre-fetching such lock(s) is/are wasted and introduce additional ldlm blocking callback RPC(s), maybe cause the performance worse. So some new mechanism should be introduced to allow server to tell client that to adjust its aggregate degree or its prefetching behavior. A simple way for that is to return some flags in the aggregate stat RPC reply.

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.