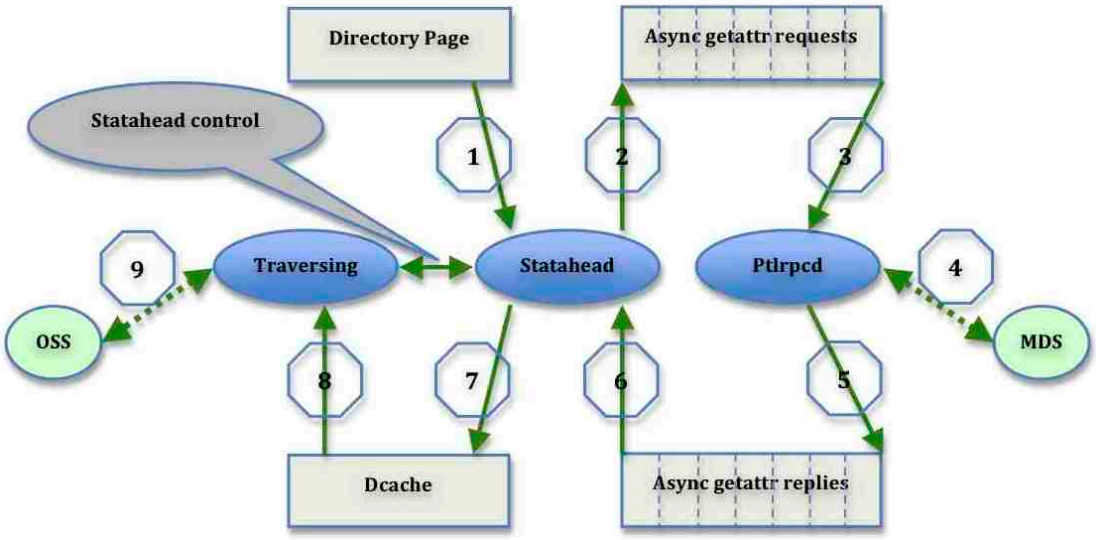
Statahead and AGL

Traversing directory sequentially, like “ls –l”, “du”, “find”, are often used system commands. To process these commands, Lustre client will trigger several RPCs (lookup/getattr, getxattr, glimpse size, and etc) for each file under the directory. Originally, these RPCs were serial and synchronous, so the performance was bad. It is quite necessary to improve traversing directory sequentially to make these frequently used system commands to run efficiently.

1. Background

In fact, to improve the performance of traversing large directory sequentially, Lustre introduced statahead since 2007. This is the outline for original statahead in Lustre:



As shown, there are three blue color threads: traversing thread, statahead thread and portal RPC daemon. The process is like this:

- At the beginning of traversing directory, the traversing thread activates the statahead thread.
- Then the statahead thread pre-fetches directory pages, scans each item in them, and pushes asynchronous getattr RPCs into the queue for MDS side attributes.
- And then the portal RPC daemon scans such queue, and sends them to MDS one by one without waiting.
- On MDS side, it handles these asynchronous getattr RPCs, grants locks and returns related attributes to client.
- For the statahead thread, it also scans asynchronous getattr RPC reply queue, processes them one by one, builds related inode and dentry, aliases them, and modifies dcache.
- With these cached information, the traversing thread can find what it wants locally without communication with MDS.

Because the three threads can run in parallel, most of the time for handling synchronous getattr RPCs are hidden by the asynchronous getattr RPCs pipeline, so the performance is improved.

Generally, the original statahead could work, but there were some defects in original design: when statahead thread processes asynchronous getattr RPC reply, it builds related inode/dentry, aliases the dentry with the inode, and adds the dentry into dcache. But at that time, it maybe conflict with other VFS operations, such as create, and confuse the dcache and other VFS operations. It is difficult to control such race condition. So we have to disable statahead by default on Lustre-1.8.

On the other hand, please note the lower-left corner of the schematic: for regular striped file, after finding the dentry/inode, the traversing thread will trigger synchronous glimpse size RPCs to OSTs for size information. Original statahead doesn't cover these RPCs. It should be improved.

2. Functional specifications

We want to resolve two main issues for improving the performance of traversing directory sequentially: one is the race condition between statahead thread and other VFS operations, the other is to replace synchronous glimpse size RPCs from client to OSTs by pipeline asynchronous glimpse lock RPCs.

2.1. Statahead local dcache

The main purpose of statahead is to save the RPCs (to MDS) of traversing thread by pre-fetching files attributes (by statahead thread) from MDS asynchronously. These attributes are related with file's inode, not dentry. But only pre-fetching files attributes is not enough, it should make these pre-fetched files are visible to the traversing thread. For that, the statahead thread builds the dentry, aliases the dentry with the inode, and adds the dentry into dcache. But the statahead thread breaks some VFS lock/semaphore mechanism when aliases the dentry with the inode, and adds the dentry into dcache. Because, to avoid possible deadlock between the traversing thread and the statahead thread, the statahead thread cannot obtain some parent's lock/semaphore. That is why conflict with other VFS operations.

In fact, there is another way to make these pre-fetched files visible to the traversing thread: statahead local dcache.

When traverses directory sequentially, the statahead maintains a small cache against the parent directory. For each pre-fetched files, the statahead thread builds/updates related inode, and records the inode together with the file's name and ldlm lock handle (without holding ldlm lock reference) as one item in the statahead local cache (for multi-linked file, there are multiple items for each

<name, inode, handle> triples). The statahead thread neither builds dentry, nor aliases dentry with inode, has nothing related with global dcache.

When the traversing thread lookup/revalidate some dentry, it will search parent directory's statahead local dcache. If finds, the traversing thread verifies whether the cached ldlm lock handle is still valid or not. If yes, then can alias the dentry (created by VFS) with the inode and adds/rehashes the dentry in global dcache. Since the traversing thread triggers lookup/revalidate through VFS layer, it must follow VFS lock/semaphore mechanism, and will not conflict with other VFS operations.

The statahead local dcache only exists during the traversing directory sequentially. It is initialized when traversing started, and finalized when traversing finished.

2.1.1. Statahead windows size

Means how many files can be pre-fetched by the statahead thread ahead of the traversing thread. We do not want to per-fetch too much files, because statahead is based on prediction. If prediction is wrong, then pre-fetched files are unless. On the other hand, more pre-fetched files means more ldlm locks cached on client, once the cached ldlm locks count exceeds some limit, some pre-fetched ldlm lock (but not used by the traversing thread) will by dropped by the client. The statahead windows size is adjustable. The proc interface "**statahead_max**" is used for that. You can modify the statahead windows size by:

echo xxx > /proc/fs/lustre/llite/${FSNAME}/statahead_max

or

lctl set_param -n llite.${FSNAME}.statahead_max xxx

If "**statahead_max**" is 0, then statahead will be disabled. Otherwise, the max statahead windows size will be "xxx". The default value is 32.

2.1.2. Query efficiency

The statahead thread is the local dcache producer, for each asynchronous getattr RPC reply, a cache item is added into the statahead local dcache. The traversing thread is the statahead local dcache consumer, after each lookup/revalidate cache hitting (as long as name matched, then means the prediction of traversing behavior is right, in spite of cached ldlm lock handle valid or not), corresponding cache item is deleted from the statahead local dcache. So the statahead local dcache size depends on the real statahead windows size. It will almost keep the same size as the real statahead windows size. So by default, the statahead local dcache size is very small. On the other hand, to improve the performance of searching in the statahead local dcache, the statahead local dcache is organized as 32 hash buckets. So normally, under default case, the average compared items count for searching in the statahead local dcache is about 1, quite efficient.

2.2. Asynchronous glimpse lock

To hide the time of handling synchronous glimpse size RPCs for traversing directory sequentially, we introduce asynchronous glimpse lock RPC to pre-fetch file size from OSTs, called AGL. The basic idea for AGL is as following:

- At the beginning of traversing directory, the AGL thread is activated.
- When the statahead thread builds/updates inodes and adds them into the statahead local dcache, for regular striped files, it also pushes their inodes into the AGL pipeline.
- The AGL thread scans the AGL pipeline, for each item in it, if no cached lock, pushes asynchronous glimpse RPCs into the queues for size/blocks information ahead of the traversing thread using it.
- And then the ptlrpcd threads scan their queues, and send AGL RPCs to OSTs asynchronously without waiting.
- For OST, if there is no conflicting lock held by other, it will grant lock to client together with size information; otherwise no glimpse callback to the conflicting lock holder, the traversing thread will process this case by itself later.
- On client side, if glimpse lock is granted, then caches size information. Otherwise, if OST does not grant lock, or the granted lock is cancelled before the traversing thread using it, then the traversing thread will trigger normal synchronous glimpse size RPCs as original stat does.

To make sure AGL doesn't change original glimpse semantics, once conflicting lock held by other, related AGL RPC is useless. Such additional AGL RPCs may affect the performance a bit, but AGL works well for large directories, since stat/read cases are more common than writes.

2.2.1. LDLM_FL_BLOCK_NOWAIT

Normally, for the synchronous glimpse size RPC, if OST finds someone holds the conflicting lock against the request, it will send glimpse callback RPC to the conflicting lock holder to query the latest size information. For the client holding the conflicting lock, when it receives the glimpse callback RPC, it will return the latest size information, and if the conflicting lock has not been used for some time (10 seconds), it will release such lock.

But for AGL case, above glimpse callback mechanism does not work. Similar as statahead, AGL is based on prediction, it may be wrong. AGL should guarantee the size information cached on the client is valid, in spite of whether the size user is the traversing thread or not. There is an uncontrolled interval between the AGL RPCs replied and the size user accessing size information. If no related ldlm lock granted and cached, the size user cannot believe the size on client. The size user has to fetch size information from OSTs by itself. So for AGL, if OST finds conflicting lock held by other, sending glimpse callback to the conflicting lock holder cannot help much, because the holder may be still using the lock, or the lock unused time is less than 10 seconds.

The AGL RPC set "LDLM_FL_BLOCK_NOWAIT" flag to tell the OST that if someone holds conflicting lock against the AGL request, then do not send glimpse callback to the conflicting lock holder.

2.2.2. CLIO lock state machine changes

To support the AGL thread to dispatch AGL RPCs ASAP, we need to adjust CLIO lock state machine as following:

- Allow to call unuse() against the cl_lock in "CLS_ENQUEUED" state.
  For each striped file, the AGL thread just pushes the AGL RPCs into related ptlrpcd threads sets, without waiting for the RPCs replies. When the AGL thread exits CLIO lock processing for some file with unuse() called, related AGL RPC may be not sent/replied yet, and related cl_lock is in "CLS_ENQUEUED" state.
- Re-trigger glimpse size RPC against the cl_lock in "CLS_ENQUEUED" state.
  As described above, for OST, if someone holds conflicting lock against the AGL request, the OST will not send glimpse callback to the conflicting lock holder. It is the traversing thread duty to process such case. When the traversing thread accesses some file size/block information, it maybe find the cl_lock (created by the AGL thread) in "CLS_ENQUEUED" state, then the traversing thread needs to wait() the AGL RPC reply. In the wait() processing, if the AGL RPC is replied without glimpse lock granted by the OST, then the traversing thread re-triggers normal glimpse size RPC to the OST against the cl_lock in "CLS_ENQUEUED" state.
- OSC lock upcall for AGL
  Because the AGL thread maybe exits CLIO lock processing before the AGL RPC reply, it is necessary to hold user reference count on the cl_lock to prevent to be cancelled/deleted by unuse() when the traversing thread exits CLIO lock processing for the file. For normal glimpse size RPC, the OSC lock upcall will signal the RPC sponsor to change the cl_lock state. But for AGL case, its sponsor (the traversing thread) maybe exits the CLIO lock processing already. So original signal mechanism maybe not work. So the OSC lock upcall will call unuse() to change the cl_lock state according to the RPC reply (glimpse lock granted or not) and release the user reference count.

2.2.3. AGL control interface

The AGL can be controllable by the proc interface "**statahead_agl**" as following:

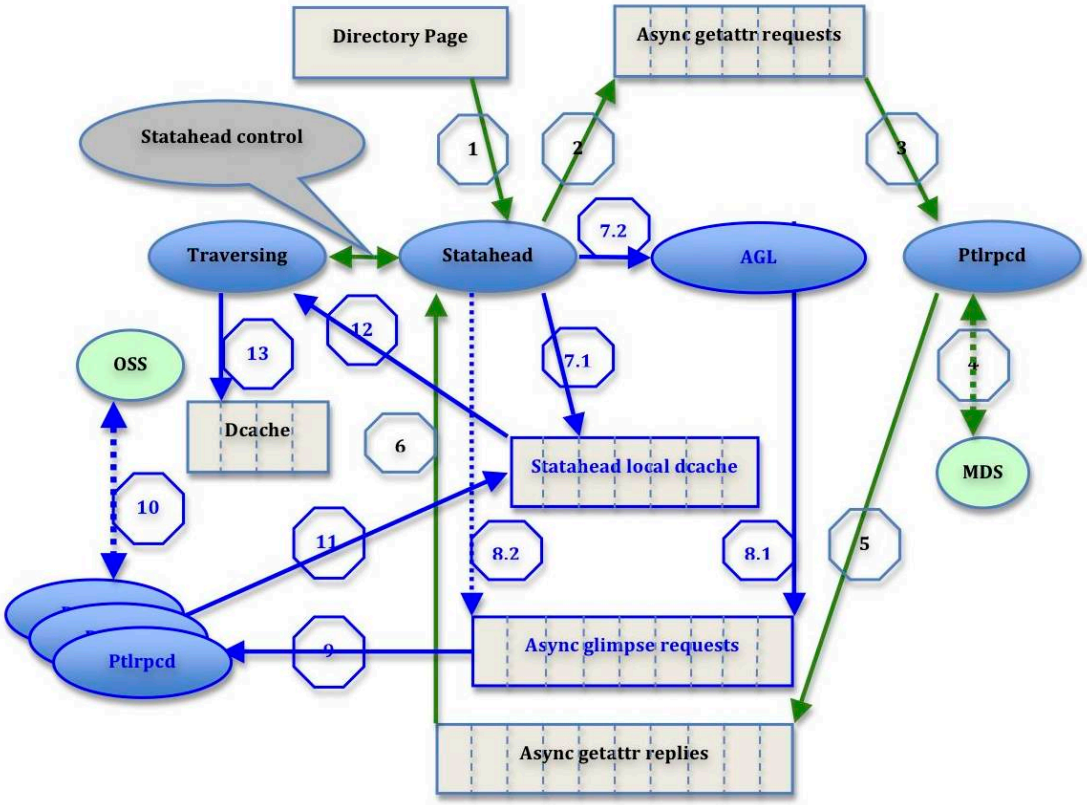
echo N > /proc/fs/lustre/llite/${FSNAME}/statahead_agl

or

lctl set_param -n llite.${FSNAME}.statahead_agl N

If “N” is zero, the AGL is disabled; otherwise the AGL is enabled. On the other hand, AGL is affected by statahead. Because the inodes processed by AGL are build by the statahead thread. That means the statahead thread is the AGL pipeline input. So if statahead is disabled, then the AGL is disabled by force.

2.3. Outline for new statahead and AGL



As shown above, the elements marked as right blue color are new introduced.

2.3.1. Double pipelines

Before introducing AGL, there is only single pipeline between the traversing thread and the statahead thread to accelerate traversing directory sequentially. But now, there are two pipelines for the accelerating:

- Statahead pipeline
  Between the traversing thread and the statahead thread, driven by the statahead thread for asynchronous getattr RPCs.
- AGL pipeline
  Between the statahead thread and the AGL thread, driven by the AGL thread for asynchronous glimpse lock RPCs.

Ideally, the double pipelines run fully, and drive low layer multiple ptlrpcd threads efficiently. Then the traversing thread, the statahead thread, the AGL thread, and multiple ptlrpcd threads can run in parallel, most the time of handling synchronous getattr RPCs and synchronous glimpse size RPCs are hidden by the double pipelines. So the performance of traversing directory

sequentially is much improved. As for the detailed test results, please refer to the presentation "Lustre Metadata Performance Improvements", which was made at LUG 2011 at Orlando.