High Level Design for Readdir+

Fan Yong

2010-12-29

0.1 Introductions

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. We have brought forward some possible solutions in another HLD named <High Level Design for Aggregate Stat>. Here, we will describe the solution of readdir+ in detailed. It can be used together with aggregate stat to improve the performance of traversing large directory.

Firstly, consider the current "ls -l" processing:

1) opendir

2) readdir for one directory page

3) lookup/getattr for one item

4) glimpse size for one item

5) repeat 3) - 4) for all items contained in the directory page

6) repeat 2) - 5) for all the directory pages

7) closedir

Assume that there is “N” pages for some specified directory “D”, and each directory page contains “M” items, and each item has only one stripe on the same OST, then on a clean client (without any cache), the total RPC count for “ls -l D” is:

*1 (for opendir) + N (for readdir) + M * N (for lookup) + M * N (for glimpse size) + 1 (for closedir)*

Consider statahead effect, some readdir RPCs and lookup RPCs are overlapped with glimpse size RPC, so the non-overlapped RPC count is not less than:

*1 (for opendir) + 1 (for readdir) + 1 (for lookup) + M * N (for glimpse size) + 1 (for closedir)*

If we introduce aggregate stat, assume the aggregation rate is “M” (the same as item count in each directory page), and there is no conflict modification operation when traversing such directory “D”, then the non-overlapped RPC count is not less than:

> *1 (for opendir) + 1 (for readdir) + 1 (for aggregate getattr) + N (for aggregate glimpse size) + 1 (for closedir)*

It is time to describe readdir+. The basic idea for readdir+ is that returning items' attributes from MDS together with directory page(s) content when readdir, no need additional lookup/getattr RPCs as described above. So combined with aggregate glimpse size, the non-overlapped RPC count is not less than:

> *1 (for opendir) + 1 (for readdir+) + N (for aggregate glimpse size) + 1 (for closedir)*

As you can see above, the RPC count under different features is less and less, then the total latency will be less and less, so the performance is improved more and more. Maybe you have noticed that, the non-overlapped RPC count for readdir+ case is only one less than aggregate getattr case, then is it worth to do that? We will discuss this issue in later section(s).

0.2 Requirements

Because readdir+ is only optimization for RPC between client and MDS for traversing directory, but at least half of the RPC time consumed by OST side glimpse size RPC. To measure the improvement through readdir+, just skip glimpse size RPC, the simplest way is traversing directory that only contains 0-striped items. Based on that, if no other modification operations under the same directory, the performance of traversing large directory (contains 1M 0-stripe items) under high latency (> 1 ms) network environment should be 100% improved at least, compared with normal statahead mode. And more latencty more improvement. Even if on local network, there should be about 10% ~ 20% improvement.

0.3 Logic specifications

For traversing directory, as described above, we will readdir page by page in the directory, and stat item by item in the pages, most of RPCs triggered by traversing directory thread are for stat operations. Reducing such RPC count is effective way to improve the performance, aggregate stat is one, and readdir+ is another one.

For traditional readdir, MDS only returns directory content, including name, identifier, offset and so on for each item, without items' attributes. But for readdir+, besides these directory content, the items' attributes, including mode, owner, flags, ACL, stripe information and etc are all packed back together. Then no need later lookup/getattr RPC for each item in the directory pages.

Currently in lustre, when lookup/getattr an item, the attributes of such item will be returned from MDS together with related ldlm lock, with such lock granted to the client, related file attributes can be cached on the client, and maybe reused by others on the same client in future. If there is any conflict operation on MDS, such lock will be canceled/revoked by explicit RPC from MDS. On the other hand, if there are so many cached locks on client as to exceed some specified limitation, then some of such locks will be released by client automatically through aggregate RPC or binding with other RPCs. The issue of caching lots of locks on client it not simple client memory pressure, but more pressure for MDS, because MDS has to record all those locks for all clients. Consider traversing large directory, before the cached attributes for former items are reused, related cached locks are much likely dropped by the client because of more locks granted for later items. If there are seldom modification operations under the same directory when traverses, then those ldlm locks granted to each item are meaningless.

So it is a serious issue: whether grant related ldlm lock to each item for readdir+, just as lookup/getattr does for traditional readdir? If yes, it is almost the same as aggregate getattr. Here we want to explore some lockless mode readdir+ or some simplified lock mode readdir+.

0.3.1 lockless mode readdir+

So-called lockless readdir+ means no locks are granted for readdir+, only items with their attributes are returned by MDS. Then client builds dentry/inode with those attributes without any ldlm locks protected. If someone from other client modifies some item(s) under the same directory, according to current consistency mechanism, if the client caches related attributes fetched through readdir+ without any related ldlm locks held, then MDS will not notify the client such change(s), and then related dentry/inode on such client are staled, but the client knows nothing yet.

Readdir+ implies pre-fetching. If without synchronization control between traversing thread and other modification thread, then even if the traversing application sees stale attributes because of above race case, it can be explained as that the traversing occurs before the modification, `which` is reasonable under distributed environment. For most traversing cases, they only run individually without synchronizing with others, and only use those kernel-cached attributes once. So above race condition is not serious for those traversing directory applications.

But from the kernel view, it is difficult to know exactly whether it is traversing directory application or not. All the pre-fetched attributes by readdir+ are prediction-based. If it is wrong, and if there is cooperation between the application on client1 and the modification application on another client2, then it maybe not the expected result of seeing stale attributes on client1. For example:

1) thread1 on client1 calls readdir, it triggers readdir+ RPC, and obtains item1's attributes without ldlm lock held on client1
2) thread1 notifies thread2 on client2 to modify item1's mode
3) thread2 on client2 modifies item1's mode as thread1 told

> *4) thread2 notifies thread1 that the modification is done*
> *5) thread1 stats item1 to check its mode, what the expectation is the new mode, but stale one is returned.*

The result of such wrong prediction is irreversible. Since lustre cannot restrict application's behavior mode, it must guarantee its prediction is unharmed for the system correctness. There are two directions to resolve such issue: one is making application to tell kernel its behavior mode explicitly, the other is simplified lock mode readdir+.

0.3.2 lustre special traversing directory tool

Since prediction-based lockless mode readdir+ maybe cause irreversible result, we can replace the prediction with notification by application explicitly. That means we will supply some special lustre tool for traversing directory. The simplest way for that is wrapping the system tool "ls" with some special lustre ioctl operation to tell kernel that it is lustre traversing directory operation (something like "lfs list"), and lockless mode readdir+ is preferred. Although not all the traversing directory applications are benefit from that, it supplies a way to traversing large directory efficiently. Even if it is not the best choice for improving the performance of traversing large directory (especially under high latency environment), the lustre special traversing directory tool can be an alternative one, particularly when many concurrent modification operations under the same directory.

Consider the following operations sequence on two race clients, assume dir1 only contains file1 (-rw-r–r– 1 root root 0 Dec 7 15:57) and file2 (-rw-r–r– 1 root root 0 Dec 7 15:57 file2):

> *1) thread1 on client1 runs "ls -1 dir1", it has finished readdir, but not stat file1 yet*
> *2) thread2 on client2 runs "touch dir1/file0"*
> *3) thread2 on client2 runs "chmod 0444 dir1/file1"*
> *4) thread1 on client1 continues its "ls -l dir1" with stat file1*

At last, the output of thread1 on client1 is as following:

> *total 0*
> *-r–r–r– 1 root root 0 Dec 7 15:59 file1*
> *-rw-r–r– 1 root root 0 Dec 7 15:57 file2*

For thread1, it did not saw the result of 2) "touch dir1/file0", but it saw the result of 3) "chmod 0444 dir1/file1". From the view of thread2, step 2) really occurred before step 3). So it is confused, seems wrong behavior/result for thread1. Currently, lustre cannot prevent that, and there is no semantic standard in POSIX for that. Means even on local filesystem, similar situation maybe occurs between two race threads. If we exchange the order of step 3) and step 4), the output of thread1 on client1 should be:

total 0
-r–r–r– 1 root root 0 Dec 7 15:57 file1
-r–r–r– 1 root root 0 Dec 7 15:57 file2

But in fact, file1's mode maybe already changed just between "stat file1" and "stat file2", means the result of "ls -l dir1" is staled already when it prints the output, it is just a snapshot of some past time-point. So what I want to say is that the output of thread1 on client1 is quite uncertain, depends on the race conditions, and the output does not guarantee its validity. It breaks nothing that we use lockless mode readdir+ to fetch bulk items' attributes and cache such attributes just for lustre special traversing directory tool itself for very short time (the interval between readdir+ and stat).

On the other hand, since MDS object attributes can be snapshot without related ldlm locks cached on client, then the same case for glimpse size attributes from OST. We can use lockless mode aggregate glimpse RPC to fetch items' size/xtime attributes from OST(s), and make them just useful for lustre special traversing directory tool.

0.3.3 simplified lock mode readdir+

This is the key topic. We want to introduce some lightweight lock mechanism to replace the original mode of one ldlm lock protecting one item's attributes for traversing directory. In fact, for a huge directory, most accessing of its content are read modes, and modification operations are relative seldom. The similar situation for accessing its children, most cases are reads, seldom ones are modification operations. Based on such assumption, we can introduce a new ldlm lock on the directory object to protect readdir+ and some specified modification operations under the same directory. Call it as LIST lock temporary, some similar as subtree lock, but not the same.

0.3.4 LIST lock policy

LIST lock works on the directory object, the protected is not the directory itself, but are its children items. That is different from lookup/update/open ldlm lock. It does not distinguish which child item to be protected, any specified modification operation on any item under the same directory will cause LIST lock conflict and LIST lock cancel. It is not all modification operations under the same directory will cause readdir+ result cached on client to be staled, like mknod, it just adds a new object under the directory, which maybe seen by traversing thread on client, maybe not, both cases are reasonable. Currently, the following modification operations maybe cause readdir+ result cached on client to be staled:

- rmdir

  If someone removes some sub-directory just after the client readdir+ RPC returns, and before the client stats such sub-directory, then the client should get "-ENOENT", but not the stale information.

- link

  It is some complex for LIST lock. Because different items with different names (under the same directory or not) can link to the same object, any new link to such object will change the link count again, which will cause all the items' attributes with different names (under the same directory or not) linked to such object to be staled. For lustre-1.x, MDS does not know which items link to the same object, so under such case, it cannot know which clients are caching stale attributes. That is terrible. We cannot implement LIST lock on lustre-1.x to protect those multiple-linked objects, so readdir+ will not return attributes for such multiple-linked objects in lustre-1.x. As for lustre-2.x, it is relative easy, because at MDD layer, all the items linking to the same object have been recorded together with their parent directory object information, with those information, MDS can know which clients maybe cache stale attributes. The subsequent design will be based on lustre-2.x. More discuss about multiple-linked issues will be done in later section(s).

- unlink

  For single linked object, it is similar as rmdir; for multiple-linked object, it is more likes link.

- setattr

  If someone setattr on some child item just after the client readdir+ RPC returns, and before the client stats such item, then the attributes for such item cached on the client is staled. For multiple-linked object, it is similar as link case, all the items linking to the same object (setattr target) are affected.

- setxattr

  Similar as setattr.

- rename

  For source item, it is similar as setattr case. As for the target item, if it exists, then similar as unlink case.

This table shows the relationship between operation and LIST lock mode:

operation	LIST lock mode
readdir+	PR on parent
rmdir	CW on parent
link	CW on each parent which children link to the object
unlink	CW on each parent which children link to the object
setattr	CW on each parent which children link to the object
setxattr	CW on each parent which children link to the object
rename	CW on each parent which children link to the source object or target object if exists

As you can see, all the modification operations hold CW mode LIST lock on parent object, because multiple modification operations on different items under the same directory can be concurrent. If there are any conflicts between those modification operations, the original ldlm mechanism can guarantee the semantic clear.

0.3.5 Protocol changes

1. Introduce new connection flags "OBD_CONNECT_READDIRPLUS" for the consultation between client and MDS for supporting readdir+ on this connection. If support, MDS will will tell client the max buffer size for readdir+ result.

2. Introduce LIST lock, a new ldlm IBITS lock as described above, which is identified with "MDS_INODELOCK_LIST" and follows the existing ldlm IBITS lock infrastructure mechanism.

3. Introduce new RPC named "MDS_READDIRPLUS" for readdir+ between client and MDS

   - RPC request format

     header(lustre_msg_v2)|ptlrpc_body|mdt_body

     - header: the same as other RPC
     - ptlrpc_body: "pb_opc" is MDS_READDIRPLUS
     - mdt_body: "fid1" is parent fid, "size" is reused as offset/hash of the first item to be read, "nlink" is reused as client side prepared buffer size for directory content and items' attributes.

     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 (directory content and items' attributes) for readdir+ 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|mdt_body|item_desc

     - header: the same as other RPC
     - ptlrpc_body: the same as other RPC
     - item_desc: a variable length array to describe attributes size for each item
       There are several ways to fill server side buffer for readdir+ result:
       (a) Prepare "P" pages, MDS fills these pages with "N" pairs of <item attributes> in turn. It looks simple, but for those items with large attributes (long name, full stripe, full ACL, and so on), one page maybe not enough to hold one item's attributes, and to improve buffer utilization, we have to allow some items and their attributes to cross pages. Under such case, it is difficult to add those pages into client page cache for reusing by others (like normal readdir). The layout is as following:

           <item1 attributes1>...<itemN attributesN>
           |—page1—|...|—pageM—|...|—pageP—|

(b) Prepare "P" pages, the first C (C >= 1) pages are used for directory content, just like normal readdir does (with multiple pages in single RPC), and can be added into client page cache for reusing by others. As for other pages, fill them with "N" items' attributes continuously (64bits aligned), if some items' attributes are too large, then allow them to cross pages. We prefer to use this format. The layout is as following:

<item1>...<itemN> <attribute1>...<attributeN>
|-*page1*-|...|-*pageC*-|—*page(C+1)*—|...|—*pageP*—|

The attributes layout in the page is like:

item_header|<*mdt_body*>|*[mdt_md]*|*[ACL]*

struct item_header {
__u32 ih_count;
__u32 ih_lens[0];
}

- mdt_body: "ioepoch" is reused as server side filled buffer size, "ino" is reused as directory content pages count has been returned, "generation" is reused as items count has been returned.

- 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

0.3.6 Client side logic

- apply LIST lock

  It is the statahead thread's duty to trigger readdir+ RPC, statahead thread will trace application behavior, and estimate whether it is traversing directory or not. If yes, then trigger readdir+. The process is similar as UPDATE lock on parent object for readdir. Before triggering readdir+, statahead thread will check whether holds the parent's LIST lock (PR mode) or not, if not (either it is the first readdir+ call, or former LIST lock is canceled by MDS during the traversing), enqueue it. After that, it will trigger readdir+ RPC with such lock held, just like triggering readdir RPC with parent's UPDATE lock held. After the readdir+, the LIST lock will be cached on client until MDS cancel it explicitly, or the directory is closed (more discuss for it in later section).

- conversion between readdir and readdir+

  LIST lock is coarse-grained ldlm lock, maybe cause maybe pseudo-conflict operations. For example: user1 on client1 wants to setattr item1 under directory1, it will cause the LIST lock for directory1 to be canceled on client2 (if client2 obtained such lock before), in spite of whether client2 cache item1's attributes or not. That is pseudo-conflict operation. To decrease the possibility of pseudo-conflict operation (further more, to reduce the possibility of LIST lock ping-pong caused by pseudo-conflict operation), if there is any conflict LIST lock held by other, then MDS will refuse to grant LIST lock to the client, instead, MDS will try to grant UPDATE lock to notify the client that there maybe conflict operation(s) and readdir should be used, but not readdir+. From the client view, if it cannot obtain LIST lock as expected, then it will convert to the normal readdir and statahead/aggregate getattr mode. The conversion is temporary, when next directory page, client will try with readdir+ again. More discuss for pseudo-conflict operation in later section.

- build dentry/inode against readdir+ result

  Client parses the readdir+ result and builds dentry/inode one by one. Introduce new time-stamp to record inode attributes fetched time on client, and set them as current time, as long as related LIST lock is valid (older than such time-stamp) on the client, then these dentry/inode are visible and valid to anybody.

- cache LIST lock

  From the view of LLITE, the user space traversing thread will calldown lookup/revalidate item by item, since readdir+ has fetched those items with their attributes, it just needs to check whether related attributes are still valid. Then how to?

  - If there is not parent's LIST lock cached on client, then the attributes maybe staled, needs original lookup/revalidate process (check/enqueue related LOOKUP/UPDATE lock for the item itself).
  - If has parent's LIST lock on client, but the lock is newer than the inode attributes time-stamp mentioned above, means the original LIST lock was canceled, and someone re-obtained it, then the attributes maybe staled, needs original lookup/revalidate process.
  - Otherwise, it is valid attributes and can be used by traversing thread directly.

  It fact, not only traversing thread but also other normal stat operations can use the LIST lock cached on client as described above. Means all stats under the same directory are benefit from the cached LIST lock for the parent object, in spite of whether related UPDATE/LOOKUP lock (for item itself) held or not.

- cancel LIST lock

  When client receives LIST lock cancel RPC from MDS, it marks related LIST lock as invalid, if no readers are using it, drop such LIST lock at once, and replies MDS. If someone is using such LIST lock on client, then it is the last reader's duty to drop such LIST lock if it found the LIST lock has been marked as invalid. Please notice the later case, the last reader needs not to send additional RPC to MDS to tell that the LIST lock is dropped. From the view of MDS, as long as it gets reply from client for the LIST lock cancel RPC, then it thinks that work has been done, which is some different from the original ldlm IBITS lock blocking callback. Anyway, client needs not to clean related items' attributes protected by the canceled LIST lock, it is later user's duty to check whether related items' attributes are valid or not.

On the other hand, before client triggers modification RPC, it will release/cancel related (on the target object and/or its parent object) ldlm IBITS lock(s) held by itself. But it is unnecessary for LIST lock, because client knows exactly which object(s) will be affected by the modification, it can just invalid (one simple way is updating the inode attributes time-stamp) the affected object(s), to avoid the unnecessary pseudo-conflict.

0.3.7 MDS side logic

- LIST lock for readdir+

  When receives LIST lock enqueue RPC from client, MDS performs LIST lock compatible checking with non-blocked mode. If no incompatible LIST lock held by others, then grants it. Otherwise (the LIST lock held by the modification operation from the same client are regarded as incompatible), converts to the normal processing of UPDATE lock enqueue. With the granted lock, MDS cannotify client whether readdir+ (with LIST lock granted) should be done or readdir (with UPDATE lock granted).

- LIST lock for modification operation

  For above specified modification operations, MDS will try to obtain related LIST lock(s) with CW mode, it maybe blocked. Similar but not the same as other ldlm IBITS lock compatible checking.

  - All the modification operations hold CW mode LIST lock on parent object, because multiple modification operations on different items under the same directory can be concurrent. If there are any conflicts between those modification operations, the original ldlm mechanism can guarantee the semantic clear.
  - If PR mode LIST lock is held by the client that the modification operation RPC is from, then it is regrade as compatible, because client knows exactly which object(s) will be affected by the modification, it can just invalid the affected object(s), to avoid the unnecessary pseudo-conflict.
  - For other PR mode LIST lock(s) holder(s), sends ldlm IBITS lock blocking callback RPC to cancel such lock(s). Once the cancel RPC is replied, then related lock(s) is/are canceled successfully. If MDS cannot cancel those incompatible LIST lock(s) in time, then related client(s) will be evicted.
  - For those multiple-linked objects, tries to obtain CW mode LIST lock on each parent which children items linking to the target object. As described, such linking information has been recorded by MDS in lustre-2.x.

  - MDS holding CW mode LIST lock is just for canceling related PR mode LIST lock(s) held by client(s) to notify some change(s). After the modification, it will release such CW mode LIST lock at once.

- Process readdir+ RPC

  MDS does not care about whether client has enqueued related LIST lock before readdir+, it allows client to perform both lockless mode readdir+ or lock mode one. Readdir+ is not the normal sense bulk RPC, according to above reply format, MDS does not know how many items can be returned until read those items and their attributes already. The process is as following:

  1. Allocate pageA, fill it with normal readdir locally.
  2. Allocate pageB, lookup all items in pageA in turn, fill pageB with their attributes, and update related "items_desc" with the attributes size (64bits aligned) as described above.
  3. If pageB space is not enough, but not all the items in pageA have been processed, then go to step2 for allocating new pageB.
  4. When all items in pageA have been processed, and the total used pages space is less than the max size specified by client, then go to step1 for allocating new pageA.
  5. When the total used pages space are not less than the max size specified by client, or up to the end of directory, then set "mdt_body->ioepoch" as the real used pages space size, set "mdt_body->ino" as the pages count for directory content, set "mdt_body->generation" as the items count has been processed.
  6. If there are any items in the last pageA whose attributes are not filled into the last pageB, ignore them, adjust the last processed item's record to the end of the last pageA, and update the next hash/offset to be read by next readdir+ as the first non-processed item in the last pageA.
  7. register all pageAs to LNET in turn, and then register all pageBs to LNET in turn, start bulk transfer after that.

0.3.8 OSS side logic

In fact, both LIST lock and readdir+ are not related with OSS. But consider the performance of traversing large directory, at least half of the RPC time are for glimpse size. The performance improvement from readdir+ is limited if without the optimization for glimpse size. SOM is the best solution, but do not know when it will be ready. Under such case, suggest using aggregate glimpse size RPC to optimize it.

0.3.9 Resend & replay

- There is no special processing for LIST lock resend and replay, just the same as other ldlm IBITS lock.

- If something wrong caused the readdir+ RPC reply lost, then client just resend related readdir+ RPC, no special process. And because readdir+ is idempotent operation, MDS can read the information again.
- Consider recovery, since readdir+ is idempotent operation, it needs not to be replayed.

0.4 Functional specifications

1. Administrator control interface for enable/disable readdir+

   It is proc interface named “readdirplus” on both client and MDS. Client’s “readdirplus” is for single node enable/disable readdir+. Negative means disabled by server, “0” means disabled by client, positive means enabled. It is also controlled by statahead, if statahead is disabled, then ignores “readdirplus”. MDS’s “readdirplus” is for the whole system enable/disable readdir+, “0” means disabled, positive means enabled.

2. LIST lock cancel on close

   Caching LIST lock on client after directory closed is helpful for next traversing directory (no LIST lock enqueue, some items’ attributes maybe still valid, then can save some RPCs). But LIST lock maybe cause pseudo-conflict operation, under such case, MDS has to send additional lock cancel RPC for those unused LIST locks cached on client. We allow administrator to adjust such policy according to the system real load and run situations.

   Provide a proc interface named “llcc” on client, for controlling whether cancel LIST lock when directory is closed. “0” means disabled, positive means enabled. When enabled, the LIST lock will be canceled automatically through close RPC. Otherwise, it will be cached on client even though related directory is closed, just like other ldlm IBITS lock.

3. Max link count and LIST lock

   As said above, the specified modification operations will cause related LIST lock held by client to be canceled. For the multiple-linked object, maybe need to send ldlm lock cancel RPC for all the parents that children item(s) linking to the target object, but maybe all of those canceling RPCs are unnecessary because of pseudo-conflict. If such case always occurs, the performance of traversing large directory maybe worse. On the other hand, even though in lustre-2.x, not all the linking information is recorded on MDS, once the link count is more than some limitation (defined as “LINKEA_MAX_COUNT”), then stop recording link information for such object(s). In lustre, it is unlimited for hard link count now. In the real environment, it is rare case that the link count is large than LINKEA_MAX_COUNT, but once it occurred, above LIST lock mechanism will fail. To resolve it, we can set a limitation that is smaller than

LINKEA_MAX_COUNT. Once some object's link count exceeds such limitation, readdir+ will skip related items linking to it, means only returns item(s), but without related attributes. The traversing thread or statahead thread will fetch those attributes with normal getattr RPC(s) later. Then those pseudo-conflict operations need not to send LIST lock cancel RPCs for such excessive linked objects.

4. Other statistics interfaces

   Including how many readdir+ RPCs triggered, conversion count for readdir+ to readdir, and so on.

0.5 Use cases

We can share the same test cases with aggregate stat. No repeat them here. Some additional test cases.

- pseudo-conflict test
- modification but not conflict with readdir+ test
- max link count test

0.6 discussions

This section contains some open issues.

0.6.1 aggregate getattr or readdir+

Compared with aggregate getattr, readdir+ needs less ldlm locks, reduces the overhead caused by applying and canceling ldlm locks, and reduces the memory pressure on both client and MDS, and more simple logic for statahead than aggregate getattr. That is the advantage. But on the other hand, it maybe introduces unnecessary pseudo-conflict, and then causes unnecessary LIST lock blocking callback RPCs, under such case, the performance maybe worse. Once that happened, had better to convert to aggregate getattr mode.

0.6.2 lockless mode or lock mode readdir+

Pure lockless mode readdir+ is prediction-based and maybe cause irreversible result, then maybe misguide application. I think lustre should avoid such cases. As described above, supplying lustre special traversing directory tool can avoid prediction failure.

Although other applications are not benefit from such tool, it is the most convenient and save way to traverse large directory in lustre. I do not want to replace the simplified lock mode readdir+ with such tool, just want to know whether it is worth to implement such tool additionally.

0.6.3 pseudo-conflict operation

LIST lock is coarse-grained ldlm lock, it is impossible to eliminate all the pseudo-conflict operation, unless back to the original one object one lock mode. But decreasing LIST lock grain can reduce the possibility of pseudo-conflict. Whether we can design page-grained LIST lock? just like ldlm extent lock on OST side. Consider the benefit from such improved LIST lock and the complexity introduced for that, whether it is worth to do?

0.6.4 Scalability & performance

Introducing LIST lock for readdir+ can reduce ldlm lock count for traversing directory, it is helpful for MDS ldlm maintaining, especially when many clients perform traversing directory operations. Both the performance and scalability are benefit from it. But on the other hand, once pseudo-conflict operations occurred, MDS has to send LIST lock blocking callback RPCs to those clients holding conflict LIST locks, and if maybe clients traversing the same directory, then such canceling RPCs are substantial.

So all the issues root in the pseudo-conflict, how to reduce the pseudo-conflict is the emphasis. As described in the design, when MDS found there are some specified modification operation(s) holding conflict LIST lock, it will grant UPDATE lock to those clients to tell them to use readdir, but not readdir+. Further more, if the specified modification operations occur very frequently under some directories during some period, should disable readdir+ for related directories temporary, in spite of whether there are conflict LIST locks held by specified modification operations when enqueue PR LIST lock for readdir+.