Optimized Stripe Assigment (QOS)

Nathan Rutman

4/24/06

1 Introduction

There are currently four optimizations that need to be done when deciding where stripes for new files are allocated.

1. Space remaing on each OST ("QOS" from b1_5)
2. OSS optimization, where we try to improve network usage by distributing stripes between OSS's (nodes) rather than OSTs.
3. Pools, where some files are administratively allowed only on certain OSTs
4. Extension-based assignment, where certain types of files are placed on certain OSTs (similar to pools).

This HLD concerns itself with integrating OSS optimization with QOS.

2 Requirements

OSS optimization is desireable for MountConf due to dynamic OST addition - OST indicies are assigned in the order OSTs first mount, rather than the order listed in an lmc script. Distributing stripes evenly among OSS's automatically, independent of OST index will help optimize network load.

3 Functional specification

We will add a weighting system to choose stripe indicies. Weights will be based on available space, reduced by various penalties. There will be per-OSS penalties, such that if a stripe was recently assigned to this OSS, the penalty will be higher. There will also be per-OST penalties, such that if a stripe was recently assigned to this OST, the

penalty will be higher. This will cause stripes to be spread out between OSS's, and between OST's within each OSS.

The weighting system should be fairly easy to extend to other optimizations.

The penalties are specified as follows:

OSS_penalty = sum(OSS_kbytes_avail)*(num_OSSs - counter)/num_OSTs/2

OST_penalty = OST_kbytes_avail * (num_OSTs - counter)/num_OSTs/2,

where *counter* is incremented every time stripe assignments are made for a new object.

These penalties will be subtracted from the amount of free space per OST, and a new sorted list created from these final weights. Stripe indicies will be assigned in a random order front-weighted from this list (See b1_5 alloc_qos() for this algorithm).

Additionally, under some conditions (# of OSTs < 2, net weights are all within 5% of each other), we will revert to a Round-Robin allocation, which can save list sorting and random choice steps. The Round-Robin list will be optimized for the number of OSTs on each OSS, but no other factors (e.g. space or last-allocated).

4 Use cases

Example 1: We have 65536kB avail/OST, 4 OSTs/OSS, and 8 OSSes. We allocate an object from OST #2, on OSS 'A' ("A2"), call this Allocation 0. So after Allocation 0 on A2, the penalty on A2 will become:

allocation penalty for OSTs on OSS 'A'

(4 * 65536)*(8 - 0)/32/2 = 8 * 4096 = 32MB

allocation penalty for OST #2 on OSS A

65536*(32 - 0)/32/2 = 32 * 1024 = 32MB

So at the next allocation (Allocation 1) the penalties on A2 will be 32MB for the OSS, and 32MB for the OST, so it would have a final weighting of A2 = 64MB - 32MB(OSS) - 32MB(OST) = 0MB. A1 would have 64MB - 32MB(OSS) - 0MB(OST) = 32MB. B1 would have 64MB - 0MB(OSS) - 0MB(OST) = 64MB. Sorted list would be B1-4, C1-4, ... A1, A3...A2.

Assuming that Allocation 1 comes from B3, then at Allocation 2 the penalties would be as follows:

OSS A: (4 * 65536)*(8 - 1)/32/2 = 7 * 4096 = 28MB

OST A2: 65536*(32 - 1)/32/2 = 31 * 1024 = 31MB

A2 = 64MB - 28MB(OSS) - 31MB(OST) = 5MB

A1,A3,A4 = 64MB - 28MB(OSS) - 0MB(OST) = 36MB

B1,B2,B4 = 64MB - 32MB(OSS) - 0MB(OST) = 32MB

B3 = 64MB - 32MB(OSS) - 32MB(OST) = 0MB

C1-4 = 64MB

So now sorted list is C1-4,...A1,A3,A4,B1,B2,B4,A2,B3

5 Logic specification

struct oss_data

- OSS nid uuid
- bavail
- penalty /* current penalty */
- penalty_per_obj /* how much the penalty changes after every object allocation */
- num_osts
- OSS list_head

struct lov_tgt_desc

- penalty /* current penalty */
- penalty_per_obj
- weight /* net weighting */
- oss_data *
- oss list entry

struct lov_obd

- lov_oss_list
- num_oss
- qos_dirty
- qos_dirty_rr
- qos_rr_array

during ost_add_target

1. check ltd_exp->exp_connection->c_remote_uuid against lov_oss_list
2. If not found, create a new oss_data, inc lov.num_oss
3. inc oss.refcount for cleanup
4. set lov.qos_dirty

during ost_del_target

1. dec oss.refcount
2. delete oss_data at refcount==0
3. set lov.qos_dirty
4. set lov.qos_dirty_rr

set qos_dirty in qos_update

at each alloc_qos, calc new weights:

```
        * for each oss
              o if qos_dirty
                    + add up all tgt TGT_BAVAIL to get oss_avail
                    + oss_penalty_per_obj = oss_avail / tgt_count / 2
                    + ost_penalty_per_obj = TGT_bavail / tgt_count / 2
                    + [order N(ost)]
              o decrement the OSS penalty every time: oss_penalty -= oss_ppo (with a minimu
              o [order N(oss)]
        * for each ost
              o decrement the OST penalty every time: ost_penalty -= ost_ppo (with a minumu
              o calculate the net weight per OST: ost_weight = TGT_BAVAIL - ost_penalty - o
              o keep track of max and min ost_weights
              o if max and min are within 5%, use Round Robin allocation for faster allocat
              o [order N(ost)]
        * create new sorted list based on the new net ost_weights
              o since penalties change at every allocation, there is no point in saving thi
              o [order N^2(ost)]
        * continue through existing alloc_qos() replacing refs to qos_bavail_list
    with our qos_weighted_list. Eliminate the bavail list (use an unsorted list in qos_prep
        * for each "found" ost in the weighted random allocation section:
              o reset the oss_penalty to max (num_oss * oss_ppo)
              o reset the ost_penalty to max (num_ost * ost_ppo)
        * Round-Robin allocation
              o keep this sorted list; it only changes when a new OST is added.
              o if qos_dirty_rr
```

```
+ create sorted list of osts based only on # OSTs in each OSS
      # Spread each OSS's OSTs evenly though arrayspace. Keep the OSS l
      #           0    5    0    5    0    5    0
      # sorted6 : FH   FH   FH    FH   FH    FH    (32/6=5.3), so space
      # sorted4 : FHABCFHGABFHCG  FHABCFHGABCFHG   (32/4=8), space 4-de
      # sorted2 : FHABCFHGABFHCGDEFHABCFHGABCFHGDE (32/2=16)
```

Some other general lov descriptor cleanups that should also be done:

- lov_desc is really lov settings descriptor, not status, so:
- move ld_tgt_count to lov.tgt_count (count is determined by ost_add, not by lconfig description)
- move ld_active_tgt_count into lov struct (should not part of descriptor)
- stripe_offset is unused
- ld_pattern should be ld_default_pattern

6 State management

6.1 State invariants

State will be maintained per LOV target, per OSS, and per LOV, as described in the above structures. The Round Robin sorted list is stored per LOV; potentially this will be stored per pool, when pools are implemented.

6.2 Scalability & performance

This algorithm introduces significant computational overhead at every single stripe allocation, with additional computation at every statfs call. At every stripe allocation: penalties for every OST and OSS are adjusted, checked for minimum, net weights calculated, and OST list sorted based on these net weights. Potentially, we could store the list and hope that it doesn't change much from allocation to allocation, which might reduce the average order from N^2(OST) to something less. Additionally, after every statfs call, every OSS size must be calculated. (This could be improved by storing the old TGT_BAVAIL for each target, and adjusting the OSS bavail by the delta.)

6.3 Recovery changes

None. Individual files will retain their stripe patterns in their own metadata. Optimization based on recent allocation history will simply be reset if the MDS LOV is restarted; the random and space-weighting functions will dominate at the beginning.

6.4 Locking changes

New OSTs should not be added to the LOV during the weight calculations. We can use the existing lov_lock for this purpose.

6.5 Disk format changes

None

6.6 Wire format changes

None

6.7 Protocol changes

None

6.8 API changes

None

6.9 RPCs order changes

None

7 Alternatives

Although the weighting system is good for accomodating many factors, I am concerned about the compute cycles needed.

8 Focus for inspections

Is the Round-Robin algorithm optimal for all OST configurations?

Can the efficiency of all these algorithms be improved?