Several months ago we met an issue of read performance issues (17% degradation) when working on ceph object storage performance evaluation with 10M objects (scaling from 10K objects to 1Million objects) , and found the root cause is unbalanced pg distribution among all osd disks, leading to unbalanced data distribution. We did some further investigation then and identified that CRUSH failed to map pgs evenly to each osd. Please refer to the attached mail for details.

Summary:

As mentioned in the attached mail, we proposed several possible optimization proposals for CRUSH and got some feedback from community (http://permalink.gmane.org/gmane.comp.file-systems.ceph.devel/18979). Sage suggested us take the idea of "Change placement strategy only for step of selecting devices from hosts", by adding a new bucket type called "linear", and applying a modulo-like hash function to this kind of buckets to achieve balanced distribution. We followed this suggestion and designed an **optimized CRUSH algorithm**, introduced a new bucket type called "linear" with new hash methods and an adaptive module. Please refer to the design and Implementation part for details. We also wrote some POC codes for it, please see the attached patch. And as a result, we got **more than 10% read performance improvement** using the optimized approach, we also verified that it didn't impact write performance, and for the recovery scenario, it did as well as previous "uniform" bucket.

Design and Implementation:

1. Problem Identification

1.1 Input key (pps) space of CRUSH is not uniform

Since PG# on the nested devices of a host is not uniform even if we select the device using simple modulo operation, we decide to change the algorithm of hashing raw pg to pps.

1.2 Algorithm of selecting items from buckets is not uniform

After we get uniform input key space, we should make the procedure of selecting devices from host be uniform. Since current CRUSH algorithm uses Jenkins hash based strategies and failed to reach the goal, we decide to add a new bucket type and apply new (modulo based) hash algorithm to make it.

2. Design

2.1 New pps hash algorithm

We design the new pps hash algorithm based on the "Congruential pseudo-random number generator" (http://comjnl.oxfordjournals.org/content/10/1/74.full.pdf). It defines a bijection between the original sequence {0, ..., 2^N-1} and some permutation of it. In other words, given different keys between 0 and 2^N-1, the generator will produce different integers, but within the same range {0, ..., 2^N-1}.

Assume there are np PGs in a pool, we can regard pgid (0≤pgid<2^n, np≤2^n<2*np) as the key, and then it will be hashed into a pps value between 0 and 2^n-1. Since PG# in a pool is usually 2^N, the generator just shuffles the original pgid sequence as output in this case, making the key space consisting of a permutation of {0,...,2^n-1}, which achieves the best uniformity. Moreover, poolid can be regarded as a seed in the generator, producing different pps value even with the same pgid but different poolid. Therefore, pgid sequences of various pools are mapped into distinct pps sequences, getting rid of PG overlapping.

2.2 New bucket type, Linear

We introduce a new bucket type called "**linear**", and apply a new modulo based hash algorithm to it. As the pps values assigned to each host are a pseudo-random subset of the original permutation and is possibly out of uniformity, in which situation applying modulo operation directly on integers in the subset cannot produce balanced distribution among disks in the host. To decrease deviation of the subset, we apply a balance parameter 1/balance_param to the key before conducting the modulo method.

For osd failure and recovery, it assumes that items nested in this kind of bucket will not be removed, nor new items are added, same as the UNIFORM bucket. Linear host will not introduce more data movement than the uniform bucket.

2.3 Adaptive Strategy

Since there is no one constant balance parameter applying for all cases that will result in the best PG distribution. We make it an adaptive procedure by adjusting the balance parameter automatically during the preparation for creating a new pool, according to different cluster topology, PG# and replica#, in order to gain a most uniform distribution.

1) Try different balance_param when preparing for a new pool
    - Iteratively call CRUSH(map, ruleno, x, balance_param) to get corresponding PG distribution with different balance_params
    - Calculate stdev of PG# among all osds
    - Choose the balance_param with the minimal stdev
2) Add a member variable to pool struct pg_pool_t to save the best balance_param value

The adaptive procedure can be described as following:

```
Input: cluster map, total PG number m, adaptive retry times n
Output: local optimal balance parameter balance_param
min_pg_stdev = MAX;
balance_param = a; // initial value
for trial from 0 to n {
    for pgid from 0 to m {
        calculate pps using the new generator in 2.1;
        for bucket b in cluster map // apply CRUSH algorithm
            apply corresponding bucket hashing algorithm and get a osd list for pgid
    }
    calculate pg_stdev_a by gathering all osd lists; // stdev of PG distribution among all osds
    if pg_stdev_a < min_pg_stdev {
        min_pg_stdev = pg_stdev_a;
        balance_param = a;
    }
    adjust a to a new value;
}
```

Evaluation:

1. Experiment Environment

We evaluate the optimized data distribution algorithm with ***Ceph Object Gateway*** and ***COSBench*** as a benchmark tool. Experiment cluster topology of Ceph and COSBench nodes is shown as Figure (1). Each host is attaching with 10x1TB disks as storage devices, and every of them are corresponding to a *ceph-osd* daemon, added up to 40 in total.

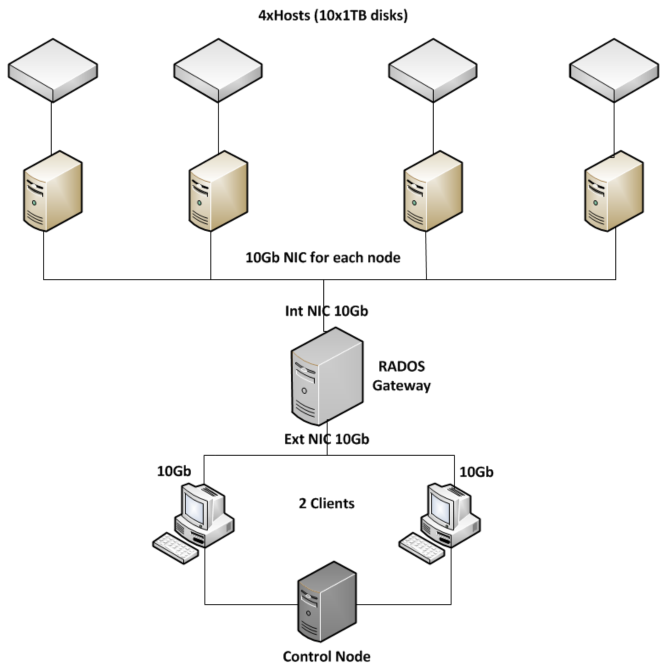


Figure (1) Ceph Cluster in Experiment Environment

2. Methodology

Testing cases and configurations are shown in Table (1).

Object size	**128KB**	**10MB**
OSD#	40	40
Pool PG#	2048	2048
Replica#	2	3
Avg. PG#/OSD	102.4	153.6
Obj#	100 million	1 million
Total data vol.	24TB	28TB
Avg. disk use%	67%	76%

Table (1) Experiment Configurations and Parameters

3. Results

Table (2) shows the performance overview of 128KB and 10MB read/write peak result case:

Bench	**# obj**	**Object-Size**	**RW-Mode**	**Throughput (op/s)**	
		--	--	**current**	**optimized**
Cosbench	100M	128KB	Read	1224.9	1388.78
			Write	1684.9	1718.05
	1M	10MB	Read	85.5	95.58
			Write	72.25	74.07

Table (2) Throughput of 128KB & 10MB read/write cases

Optimized CRUSH algorithm brings about **13%** performance improvement for 128KB read case, **12%** improvement for 10MB read case, and no sacrifice for write performance.

We also test load line performance to demonstrate the scalability of the optimized algorithm, and the results are shown in Figure 2 and Figure 3. The optimized CRUSH algorithm improves throughputs in all cases.

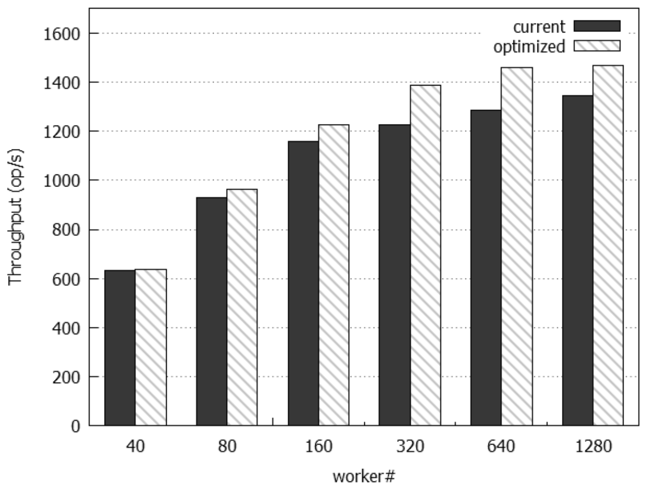


Figure (2) Throughput Load Line of 128KB Read Cases

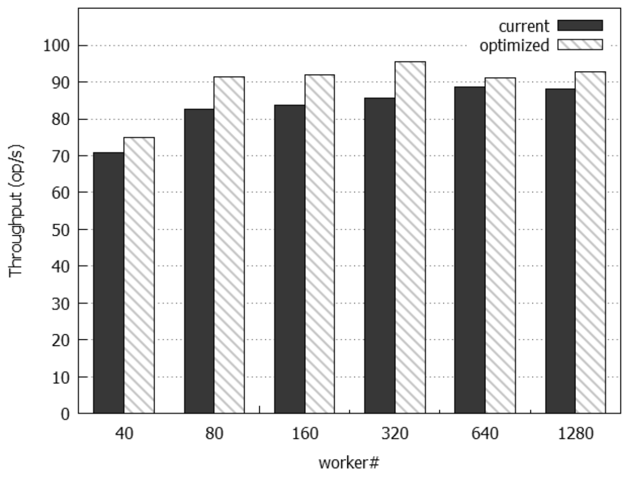
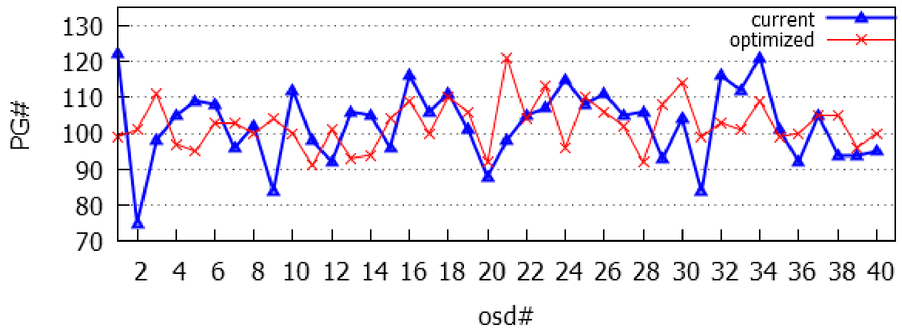
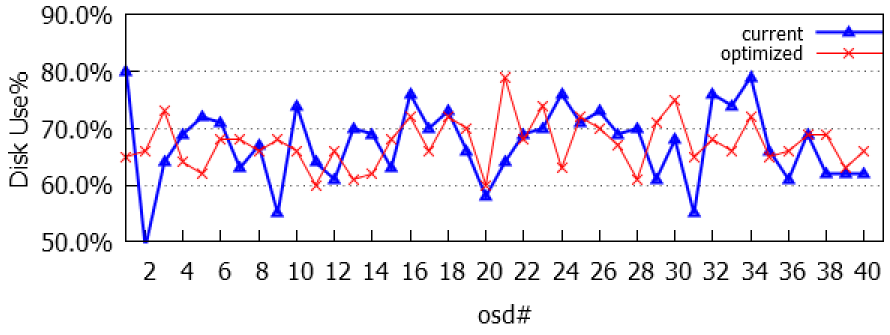


Figure (3) Throughput Load Line of 10MB Read Cases

Detailed values of PG number and disk use% on every OSD disk are represented in Figure (4) and Figure (5). For 128KB sized object cases, the optimized algorithm narrows down standard deviation of PG distribution gap **from 10.09 to 6.50**, decreased about 40%, and for 10MB sized objects cases, the gap is narrowed down **from 12.13 to 5.17**, decreased about 57%. Similar results are observed for data distribution, variation of disk use% for 100 million 128KB objects is reduced **from more than 30% to about 20%**, and that for 1 million 10MB sized objects is reduced **from more than 30% to about 10%**.

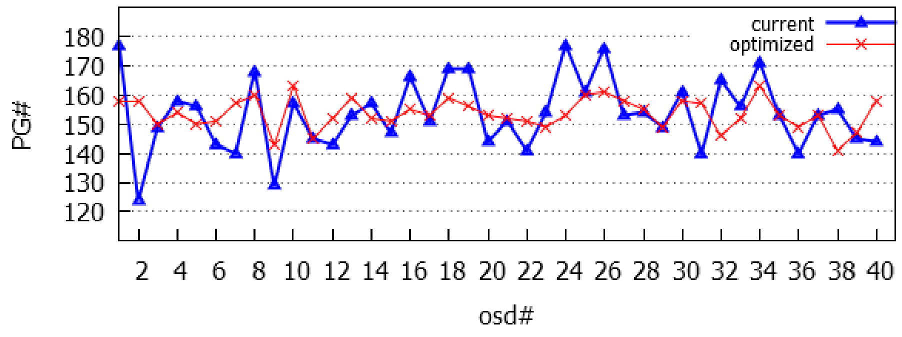


(a) PG distribution of 128KB object cases

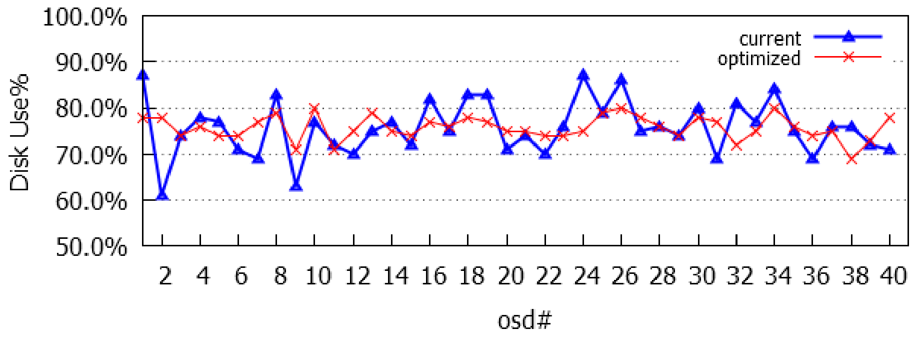


(b) Data distribution of 128KB object cases

Figure (4) PG and data distribution of 128KB object cases



(a) PG distribution of 10MB object cases



(b) Data distribution of 10MB object cases

Figure (5) PG and data distribution of 10MB object cases