Optimize Librbd for Lower CPU Cost and Higher Scalability

Li Wang
Cloud Storage Team, DiDi
Ceph Block Solution for VM

• Data flow
  – VM disk access requests are passed to *qemu-rbd*, which is a qemu backend block driver
  – *Qemu-rbd* pass requests to *librbd*
  – *Librbd* send requests over network to *Ceph*, bypass host kernel disk IO stack
Ceph Block Solution for Container

- **Data flow**
  - Container disk access requests are passed to *nbd*, a kernel block driver
  - *Nbd* forward the requests to *rbd-nbd* in user space through socket
  - *Rbd-nbd* pass requests to *librbd* to access Ceph
RBD IOPS Optimization

• Motivation
  – The rbd IOPS not scale according to our fio test, bounded at around 22K on a full SSD cluster

• Reason
  – With rbd cache on, the data are written into cache first by the thread tp_librbd, then asynchronously flushed by the thread flusher_thread
  – Both tp_librbd and flusher_thread are single thread
RBD IOPS Optimization

• Reason
  – Writing and flushing **not paralleled**
    • `tp_librbd` and `flusher_thread` are **mutually exclusive** by the cache lock
  – **Lock held throughout flushing**
    • `LibrbdWriteback::write()` consumes much time, according to CPU flame graph
    • Application writing are blocked by the lock

```c
void ImageCtx::write_to_cache(object_t o, const bufferlist& bl, size_t len,
                                  uint64_t off, Context *onfinish,
                                  int fadvise_flags, uint64_t journal_tid,
                                  ZTracer::Trace *trace) {

  ...}
  wr->extents.push_back(extent);
  { Mutex::Locker l(cache_lock);
    object_cacher->writex(wr, object_set, onfinish, trace);
  }
}

void ObjectCacher::flusher_entry()
{
  lock.Lock();
  while (!flusher_stop) {
    ...
  }
  while (reads_outstanding > 0) {
    ...
  }
  lock.Unlock();
```
RBD IOPS Optimization

• Solution
  – Enable flushing paralleled with writing
    • Move `LibrbdWriteback::write()` outside the lock
  – Improve flushing throughput
    • Enable `LibrbdWriteback::write()` done by multiple threads

• Results
  – 4K random write, IOPS reach 38K, improve by 71%

• Reference
  – https://github.com/ceph/ceph/pull/25713
Client Side Image Cache

• Motivation
  – Exploit rbd clone to save space
    • Rbd base+clone back VM system disk
    • base image store the readonly OS template data
    • all VMs with the same OS share a base image
  – Boot storm
    • hundreds even thousands of VMs of the same tenant boot at the same time
    • High IO pressure on one base image, cause a slow boot
Client Side Image Cache

• Solution
  – Store the OS template in a *qcow2* format file
  – Cache the *qcow2* file at the computing node
  – Equip *librbd* with *qcow2* parsing ability
  – When accessing base image, access the *qcow2* file locally

• Benefits
  – VM boot locally, IO pressure on Ceph cluster reduced, and boot speed improved
Librbd CPU Overhead Optimization

• Motivation
  – *librbd* cause high CPU usage 124% under *fio* 1MB sequential write
  – Much time on two memcpy
    • qemu: iov_to_buf_full
    • librbd: rbd_aio_write
Librbd CPU Overhead Optimization

• Optimize the first memcpy
  – Reason
    • Qemu `bdrv_aio_writev` call librbd `rbd_aio_write` to write data, which accepts only one input buffer
    • Qemu `bdrv_aio_writev` hold multiple buffers
    • Qemu perform a `memcpy` to collect the data from multiple buffers into a buffer to match librbd
Librbd CPU Overhead Optimization

• Optimize the first memcpy
  – Gather write
    • The librbd library installed by default is relatively old
    • Recent librbd provide another API *rbd_aio_writev*, which accept multiple input buffers
    • Upgrade the librbd, enable qemu call *rbd_aio_writev*, directly pass the buffers
Librbd CPU Overhead Optimization

• Optimize the second memcpy
  – Reason
    • Librbd call a memcpy to copy multiple buffers passed by qemu into a private buffer
    • The data are asynchronously sent to OSDs by librbd messenger
    • If possible to let librbd refer to the buffers directly, without copying data
    • The key is to make sure the buffers no longer referred by librbd after write complete
  – Race condition example
    • Librbd construct a request $x$ to send data to osd.0
    • $x$ is on the way to osd.0
    • osd.0 down and restart later
    • Librbd reconstruct a request $y$ to send data to osd.1 due to the osdmap changes
    • $y$ completes, the buffers freed by qemu
    • Messenger resend $x$ due to timeout, referring unallocated memory
Librbd CPU Overhead Optimization

• Optimize the second memcpy
  – Zero copy optimization
    • Implement `cancel_request` for messenger
    • When reconstruct request to send to new targets, cancel pending requests first
  – Limitation
    • Could not use zero copy optimization if rbd cache on with write-back mode
    • Write complete after data written into cache, when messenger refer to the buffer, it may be freed by qemu
Librbd CPU Overhead Optimization

- 1MB sequential write results
  - Latency reduce by 46%
  - IOPS increase by 85%
  - CPU usage reduce by 53%
To Discuss

- Latency
  - The latency is relatively high, especially for bluestore on SSDs
  - 4K random write, iodepth=1, the average latency is 1.8ms

- Hardware utilization
  - Filestore on HDDs, when average disk iowait reach more than 60%, tail latency, even slow request may occur, even with careful tuning of the wbthrottle parameters

- CPU usage
  - Bluestore on SSDs, high CPU usage, one SSD only contribute 11K IOPS
THANK YOU