

Ceph rados读性能瓶颈分析

本文使用《ceph性能评测方法总结》中的方法，对ceph rados层在整个读流程中的瓶颈进行分析。该分析前后在SATA SSD和NVME SSD两种存储介质中都进行过，情况相似，只是在NVME中由于硬件延迟更小，所以小粒度读中的软件问题暴露得更加明显，以下以NVME介质中的测试结果为对象进行分析。同理，OSD数量，线程数量，测试客户端压力等各种因素也都做过尝试，最后选择下述测试配置作为典型分析：单节点72cpu，3块NVME盘，30SD，每个OSD分成16个shard，每个shard两线程，客户端在起在另一节点，起5个客户端进行压测，每个客户端支持256的异步并发。

1 硬件资源使用情况分析

我们首先来看外存资源的使用情况，rados读压测时的iostat跟fio比较如下：

	r/s	r MB/s	qu-sz	await	%util
fio	388628	1554	60.29	0.16	102%
Rados	45146	180	4.32	0.10	97.32%

可以看到，即便在读客户端压力足够的情况下，外存设备的带宽也远没有被发挥出来，主要差距体现在I/O队列的平均长度上，这意味着性能差距来自于软件层不能给到下层足够的I/O压力，不能利用起NVME强大的硬件并发性能。30SD的带宽之和也远没有达到集群配置的网络带宽上限，因此也不会是网络的瓶颈。

这很容易让人觉得是cpu瓶颈的问题，那就看一下vmstat的结果，如下图所示

	run	block	in	cs	us	sy	id	wa
nvme	24	10	351482	1064607	16	6	72	6

从结果可见运行和阻塞的cpu加起来也小于核数，cpu在72%的时间里面处于较为闲置的状态，top时osd占用的总cpu也只有500%左右。

由此在硬件占用资源方面可以得到的结论是，I/O设备的资源没有用满是性能存在提升空间的主要表现，同时，在软件层面实际能够跑起的实际并发数也远小于核数，这可能是rados层的实现问题，也可能是操作系统对于多核扩展性本身就很难用好。

2 读请求各阶段耗时分析

我们使用perfcount统计读过程中各阶段耗时，结果如下：

op-r-latency: 0.25ms

含义：从收到该请求HEAD消息，到完成该请求回复客户端为止，可以理解在OSD上的整个延迟

op-before-queue-op-lat: 0.03ms

含义：从messenger收到这个读请求的HEAD开始计时，直到Messenger将该请求放入队列为止，可以理解为OSD网络模块的耗时

op-before-dequeue-op-lat: 0.1ms

含义：从messenger收到这个请求到OSD工作请求将该请求从队列中读出来为止，减去op-before-queue-op-lat就是请求在队列中的耗时，从该值来看，排队延迟不能忽略，可能原因是消费者（OSD工作线程）处在较为繁忙的状态。

op-r-process-latency: 0.16ms

含义：从工作线程将请求取出到回复客户端消息，可以理解为OSD工作线程处理一个请求的耗时。

read-lat: 0.11ms

含义：这是bluestore域的统计变量，是调用BlueStore::Read的耗时，可以看到该延迟已经非常接近硬件延迟，可见bluestore层在读时基本没有性能的浪费。

从上述结果我们可以做出的分析是，除掉必要的硬件开销耗时，占比较大的就只剩排队耗时和osd工作线程中除掉bluestore的部分

3 线程视角的统计信息

Gdbmp跟踪osd工作线程（tp-osd-tp）进行打点的结果如下（抽出了其中的核心部分）

```
code - 记事本
文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)
+ 100.00% clone
+ 100.00% start_thread
+ 100.00% ShardedThreadPool:WorkThreadSharded:entry()
+ 100.00% ShardedThreadPool:shardedthreadpool_worker(unsigned int)
+ 100.00% OSD::ShardedOpWQ::process(unsigned int, ceph::heartbeat_handle_d*)
+ 65.80% pthread_cond_timedwait@@GLIBC_2.3.2
| + 0.10% __pthread_mutex_cond_lock
| + 0.10% _l_cond_lock_789
| + 0.10% _lll_lock_wait
+ 28.80% PGQueueable::RunVis::operator()(boost::intrusive_ptr<OpRequest> const&)
| + 28.80% OSD::dequeue_op(boost::intrusive_ptr<PG>, boost::intrusive_ptr<OpRequest>, ThreadPool::TPHandle&)
| + 28.80% PrimaryLogPG::do_request(boost::intrusive_ptr<OpRequest> &, ThreadPool::TPHandle&)
| + 28.80% PrimaryLogPG::do_op(boost::intrusive_ptr<OpRequest> &)
| + 18.80% PrimaryLogPG::execute_ctx(PrimaryLogPG::OpContext*)
| + 14.80% PrimaryLogPG::prepare_transaction(PrimaryLogPG::OpContext*)
| + 14.80% PrimaryLogPG::do_osd_ops(PrimaryLogPG::OpContext*, std::vector<OSDOp, std::allocator<OSDOp> > &)
| + 14.80% PrimaryLogPG::do_read(PrimaryLogPG::OpContext*, OSDOp&)
| + 9.70% PrimaryLogPG::find_object_context(hobject_t const&, std::shared_ptr<ObjectContext>*, bool, bool, hobject_t*)
| + 9.70% PrimaryLogPG::get_object_context(hobject_t const&, bool, std::map<std::string, ceph::buffer::list, std::less<std::string>, std::allocator<std::pair<std::string const, ceph::buffer::list> > > const&)
| + 8.90% PGBackend::objects_get_attr(hobject_t const&, std::string const&, ceph::buffer::list*)
| + 8.80% BlueStore::get_attr(boost::intrusive_ptr<ObjectStore::CollectionImpl> &, ghobject_t const&, char const*, ceph::buffer::ptr&)
| + 8.80% BlueStore::Collection::get_onode(ghobject_t const&, bool)
| + 6.40% RocksDBStore::get(std::string const&, char const*, unsigned long, ceph::buffer::list*)
| + 6.30% rocksdb::DB::Get(rocksdb::ReadOptions const&, rocksdb::Slice const&, std::string*)
```

其中占28.8%的RunVis::operator是工作线程处理请求的入口函数，也就是说，工作线程处理请求的有效时间只占整个线程运行时间的28.8%，而占65.5%时间的pthread_mutex_cond显然是性能的瓶颈，那么它是如何被调用的呢？阅读源码可以发现它的调用位置如下：

```
// peek at spg_t
sdata->sdata_op_ordering_lock.Lock();
if (sdata->pqueue->empty()) {
    dout(20) << __func__ << " empty q, waiting" << dendl;
    // optimistically sleep a moment; maybe another work item will come along.
    osd->cct->get_heartbeat_map()->reset_timeout(hb,
        osd->cct->_conf->threadpool_default_timeout, 0);
    sdata->sdata_lock.Lock();
    sdata->sdata_op_ordering_lock.Unlock();
    sdata->sdata_cond.WaitInterval(sdata->sdata_lock,
        utime_t(osd->cct->_conf->threadpool_empty_queue_max_wait, 0));
    sdata->sdata_lock.Unlock();
    sdata->sdata_op_ordering_lock.Lock();
    if (sdata->pqueue->empty()) {
        sdata->sdata_op_ordering_lock.Unlock();
        return;
    }
}
```

可以看到它被调用的原因是因为操作队列为空，然后调用WaitInterval进行等待，等到操作到来时再由生产者发送信号量将其唤醒。

另外，gdbmp解释了读请求耗时中除了bluestore之外还耗时在什么地方，答案就是获得object的元数据（find-object-context）可能需要访问RocksDB，如果没有命中缓存的话会有I/O操作。

4 cpu时间视角的分析统计

在该分析中on-cpu记录的结果的参考意义不大，略过。

5 分析总结

结合perfcounter和gdbmp可以发现读在软件层的瓶颈是这样出现的：msg_r把一个请求插入队列并发送一个信号，但睡眠等待的工作线程却没能被及时唤醒起来执行该请求，导致了一方面请求需要在队列中等待一段不短的时间，另一方面工作线程不能持续执行请求，最终造成给外存压力不足，发挥不出设备的硬件性能。

当我们试图通过增加线程数来增加软件层处理请求的能力时，会发现工作线程处于睡眠阶段的时间占比随着线程数增加，如下图所示，这就是为什么增加线程数并不能显著地优化性能的原因。

工作线程数（30SD总和）	睡眠时间占比
12	22.6%
96	65.8%
384	95.4%

为什么在线程数较大的情况下，不能及时地唤醒线程，已经变成操作系统层面的问题，多核操作系统的调度复杂，也许和我们直观理解的工作方式有较大差异，只能说，传统的多线程同步I/O模型在硬件设备变得越来越强的时候已经不能很好地发挥硬件性能。但我们可以确认的是，工作线程数量较少时，线程可以拥有更高的效率，如果可以使用异步，用较少的工作线程提供较大的并发，应该可以使读性能获得明显优化。