Ceph Performance:
Projects Leading up to Jewel

Mark Nelson
Ceph Community Performance Lead
mnelson@redhat.com

4/12/2016

OVERVIEW

What's been going on with Ceph performance since Hammer?

Answer: **A lot!**

- Memory Allocator Testing
- Bluestore Development
- RADOS Gateway Bucket Index Overhead
- Cache Teiring Probabilistic Promotion Throttling

First let's look at how we are testing all this stuff...

CBT

CBT is an open source tool for creating Ceph clusters and running benchmarks against them.

- Automatically builds clusters and runs through a variety of tests.
- Can launch various monitoring and profiling tools such as collectl.
- YAML based configuration file for cluster and test configuration.
- Open Source: https://github.com/ceph/cbt

MEMORY ALLOCATOR TESTING

Example CBT Test

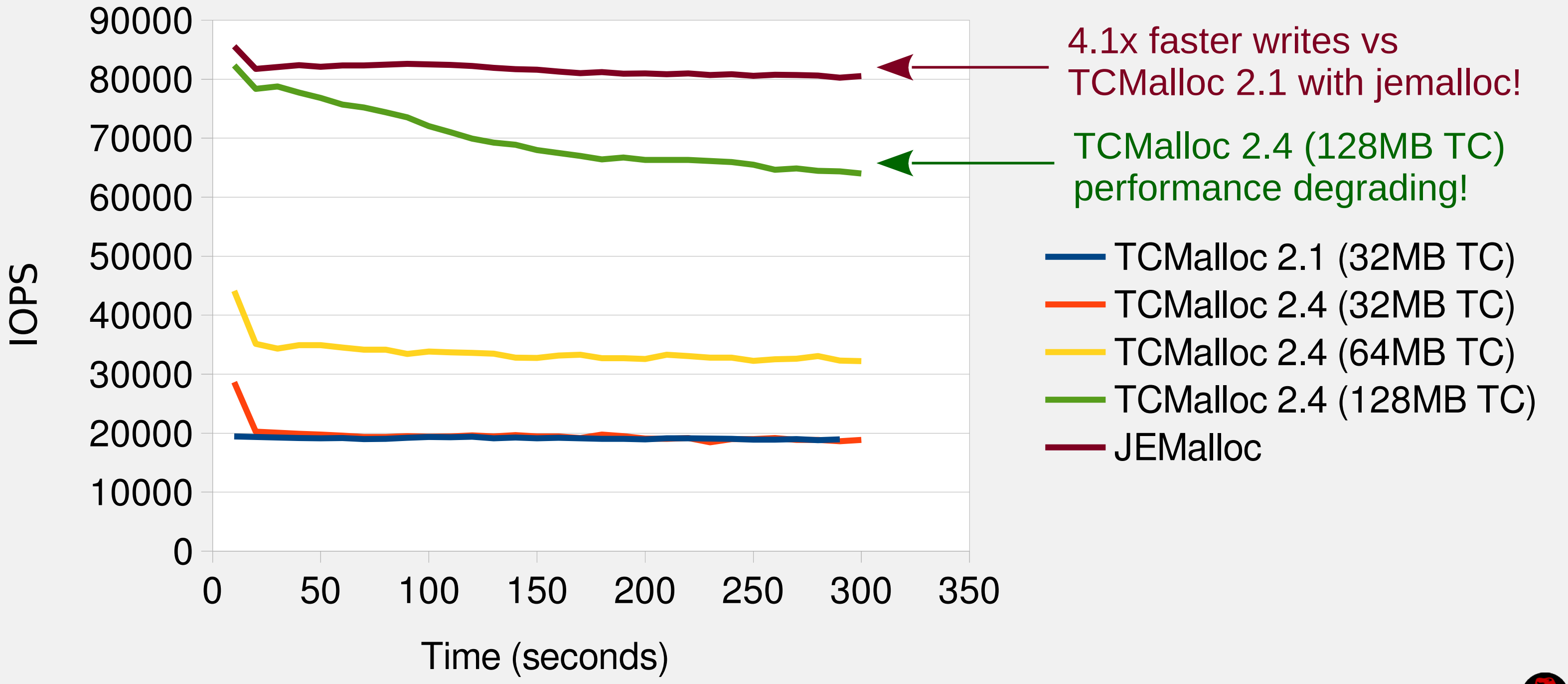
We sat down at the 2015 Ceph Hackathon and tested a CBT configuration to replicate memory allocator results on SSD based clusters pioneered by Sandisk and Intel.

Memory Allocator	Version	Notes
TCMalloc	2.1 (default)	Thread Cache can not be changed due to bug.
TCMalloc	2.4	Default 32MB Thread Cache
TCMalloc	2.4	64MB Thread Cache
TCMalloc	2.4	128MB Thread cache
Jemalloc	3.6.0	Default jemalloc configuration

MEMORY ALLOCATOR TESTING

4KB Random Writes

The cluster is rebuilt the exact same way for every memory allocator tested. Tests are run across many different IO sizes and IO Types. The most impressive change was in 4K Random Writes. Over 4X faster with jemalloc!



4.1x faster writes vs TCMalloc 2.1 with jemalloc!

TCMalloc 2.4 (128MB TC) performance degrading!

WHAT NOW?

Does the Memory Allocator Affect Memory Usage?

We need to examine RSS memory usage during recovery to see what happens in a memory intensive scenario. CBT can perform recovery tests during benchmarks with a small configuration change:

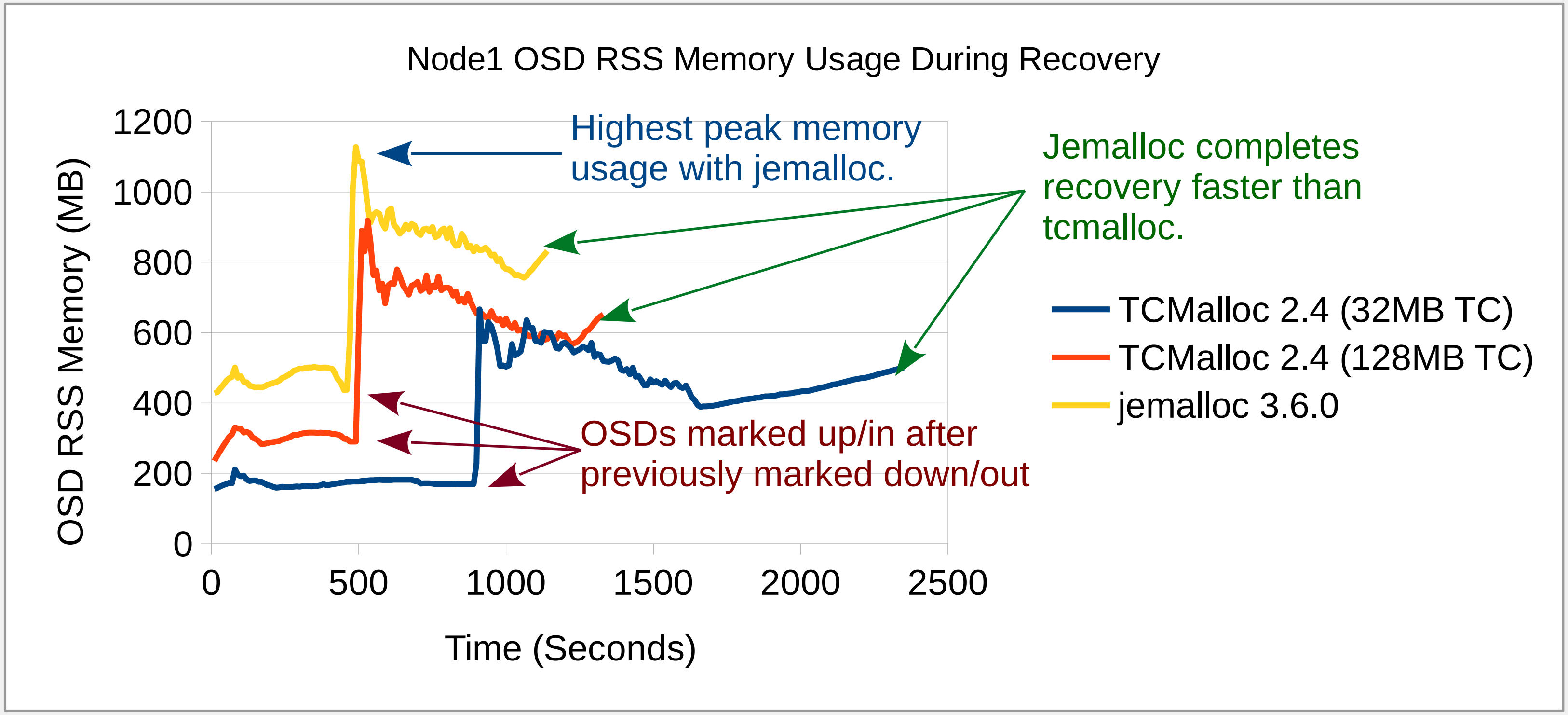
```
cluster:
  recovery_test:
    osds: [ 1, 2, 3, 4,
            5, 6, 7, 8,
            9,10,11,12,
           13,14,15,16]
```

Test procedure:

- Start the test.
- Wait 60 seconds.
- Mark OSDs on Node 1 down/out.
- Wait until the cluster heals.
- Mark OSDs on Node 1 up/in.
- Wait until the cluster heals.
- Wait 60 seconds.
- End the test.

MEMORY ALLOCATOR TESTING

Memory Usage during Recovery with Concurrent 4KB Random Writes



Highest peak memory usage with jemalloc.

Jemalloc completes recovery faster than tcmalloc.

OSDs marked up/in after previously marked down/out

MEMORY ALLOCATOR TESTING

General Conclusions

- Ceph is very hard on memory allocators. Opportunities for tuning.
- Huge performance gains and latency drops possible!
- Small IO on fast SSDs is CPU limited in these tests.
- Jemalloc provides higher performance but uses more memory.
- Memory allocator tuning primarily necessary for SimpleMessenger. AsyncMessenger not affected.
- We decided to keep TCMalloc as the default memory allocator in Jewel but increased the amount of thread cache to 128MB.

FILESTORE DEFICIENCIES

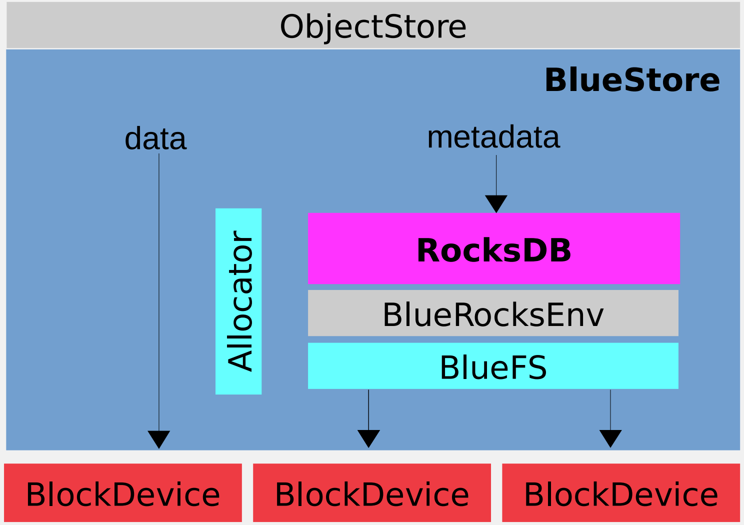
Ceph already has Filestore. Why add a new OSD backend?

- 2X journal write penalty needs to go away!
- Filestore stores metadata in XATTRS. On XFS, any XATTR larger than 254B causes all XATTRS to be moved out of the inode.
- Filestore's PG directory hierarchy grows with the number of objects. This can be mitigated by favoring dentry cache with vfs_cache_pressure, but...
- OSD regularly call syncfs to persist buffered writes. Syncfs does an O(n) search of the entire in-kernel inode cache and slows down as more inodes are cached!
- Pick your poison. Crank up vfs_cache_pressure to avoid syncfs penalties or turn it down to avoid extra dentry seeks caused by deep PG directory hierarchies?
- There must be a better way...

BLUESTORE

How is BlueStore different?

- BlueStore = **Bl**ock + **NewStore**
  - consume raw block device(s)
  - key/value database (RocksDB) for metadata
  - data written directly to block device
  - pluggable block Allocator
- We must share the block device with RocksDB
  - implement our own rocksdb::Env
  - implement tiny “file system” BlueFS
  - make BlueStore and BlueFS share



BLUESTORE

BlueStore Advantages

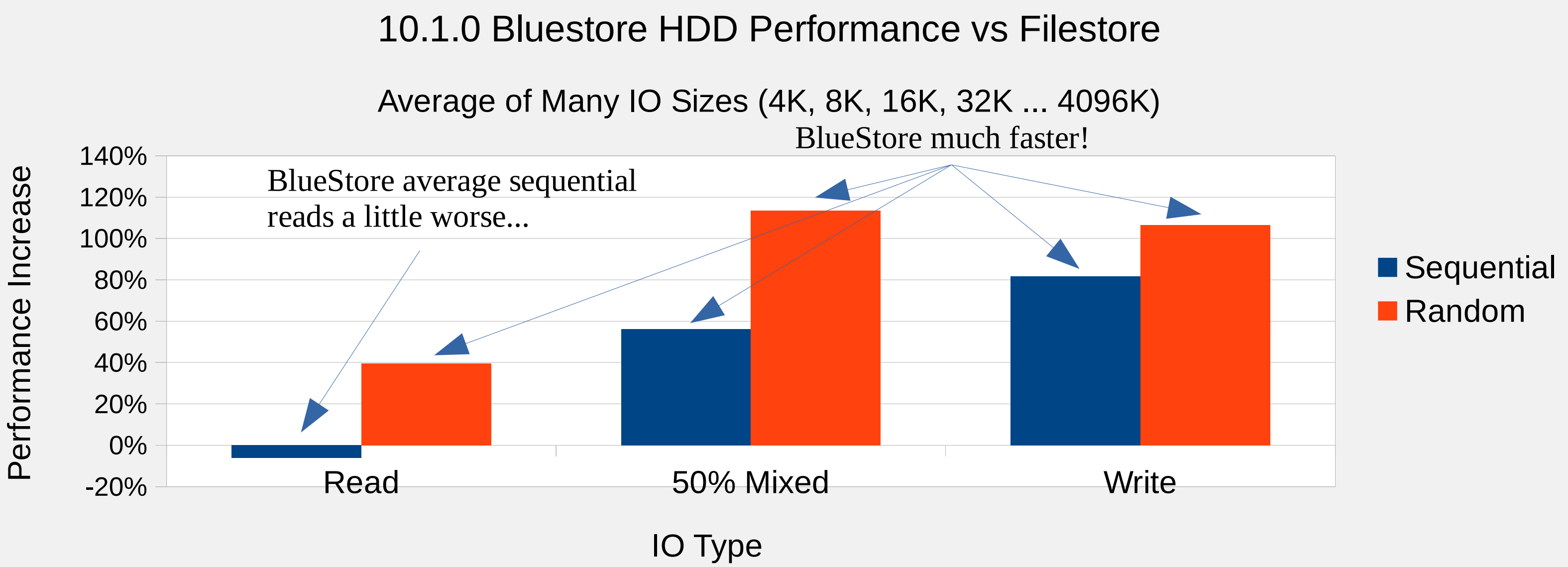
- Large writes go to directly to block device and small writes to RocksDB WAL.
- No more crazy PG directory hierarchy!
- metadata stored in RocksDB instead of FS XATTRS / LevelDB
- Less SSD wear due to journal writes, and SSDs used for more than journaling.
- Map BlueFS/RocksDB “directories” to different block devices
  - db.wal/ – on NVRAM, NVMe, SSD
  - db/ – level0 and hot SSTs on SSD
  - db.slow/ – cold SSTs on HDD

Not production ready yet, but will be in Jewel as an experimental feature!

BLUESTORE HDD PERFORMANCE

How does BlueStore Perform?

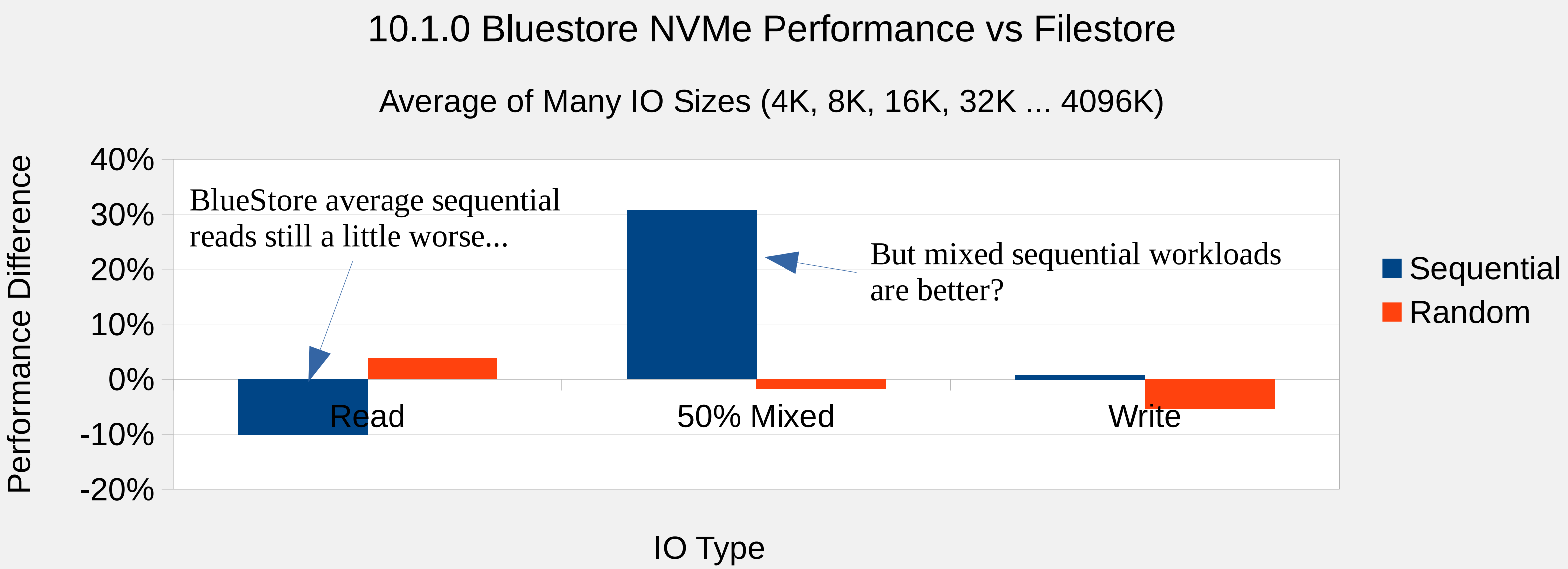
HDD Performance looks good overall, though sequential reads are important to watch closely since BlueStore relies on client-side readahead.



BLUESTORE NVMe PERFORMANCE

NVMe results however are decidedly mixed.

What these averages don't show is dramatic performance variation at different IO sizes. Let's take a look at what's happening.



BlueStore average sequential reads still a little worse...

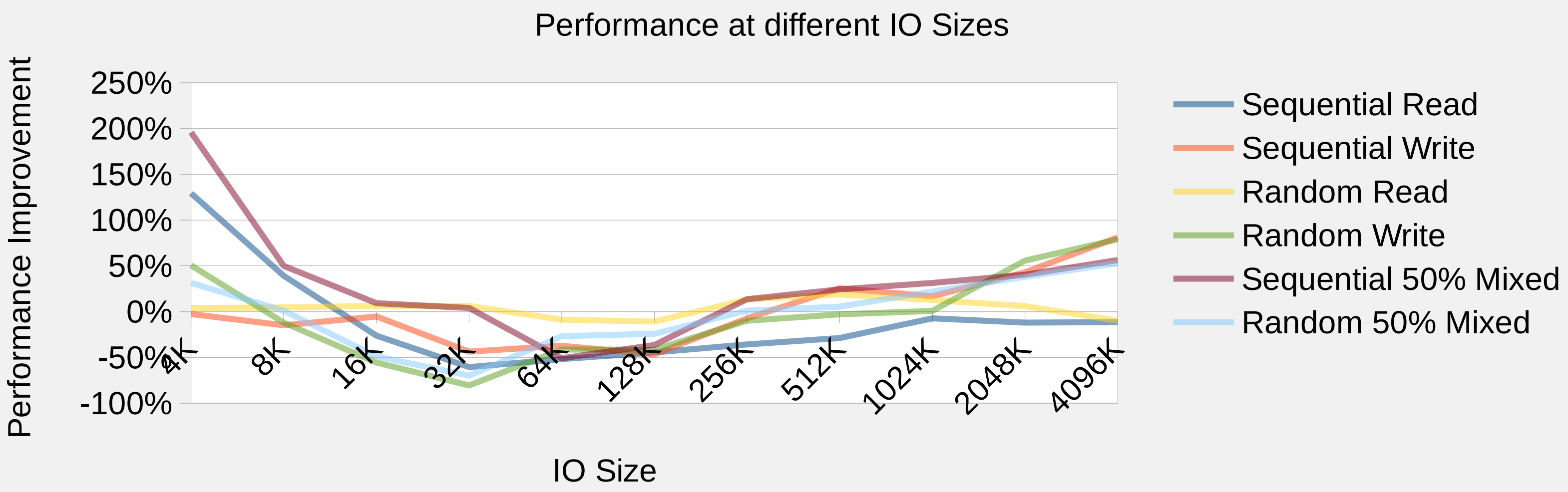
But mixed sequential workloads are better?

BLUESTORE NVMe PERFORMANCE

NVMe results are decidedly mixed, but why?

Performance generally good at small and large IO sizes but is slower than filestore at middle IO sizes. BlueStore is experimental still, stay tuned while we tune!

10.1.0 Bluestore NVME Performance vs Filestore



RGW WRITE PERFORMANCE

A common question:

Why is there a difference in performance between RGW writes and pure RADOS writes?

There are several factors that play a part:

- S3/Swift protocol likely higher overhead than native RADOS.
- Writes translated through a gateway results in extra latency and potentially additional bottlenecks.
- Most importantly, RGW maintains bucket indices that have to be updated every time there is a write while RADOS does not maintain indices.

RGW BUCKET INDEX OVERHEAD

How are RGW Bucket Indices Stored?

- Standard RADOS Objects with the same rules as other objects
- Can be sharded across multiple RADOS objects to improve parallelism
- Data stored in OMAP (ie XATTRS on the underlying object file using filestore)

What Happens during an RGW Write?

- Prepare Step: First stage of 2 stage bucket index update in preparation for write.
- Actual Write
- Commit Step: Asynchronous 2nd stage of bucket index update to record that the write completed.

  Note: Every time an object is accessed on filestore backed OSDs, multiple metadata seeks may be required depending on the kernel dentry/inode cache, the total numbrer of objects, and external memory pressure.

RGW BUCKET INDEX OVERHEAD

A real example from a customer deployment:

Use GDB as a “poorman's profiler” to see what RGW threads are doing during a heavy 256K object write workload:

gdb -ex "set pagination 0" -ex "thread apply all bt" --batch -p <process>

Results:

- 200 threads doing IoCtx::operate
  - 169 librados::ObjectWriteOperation
    - **126 RGWRados::Bucket::UpdateIndex::prepare(RGWModifyOp) ()**
  - 31 librados::ObjectReadOperation
    - **31 RGWRados::raw_obj_stat(…) ()** ← *read head metadata*

IMPROVING RGW WRITES

How to make RGW writes faster?

Bluestore gives us a lot

- No more journal penalty for large writes (helps everything, including RGW!)
- Much better allocator behavior and allocator metadata stored in RocksDB
- Bucket index updates in RocksDB instead of XATTRS (should be faster!)
- No need for separate SSD pool for Bucket Indices?

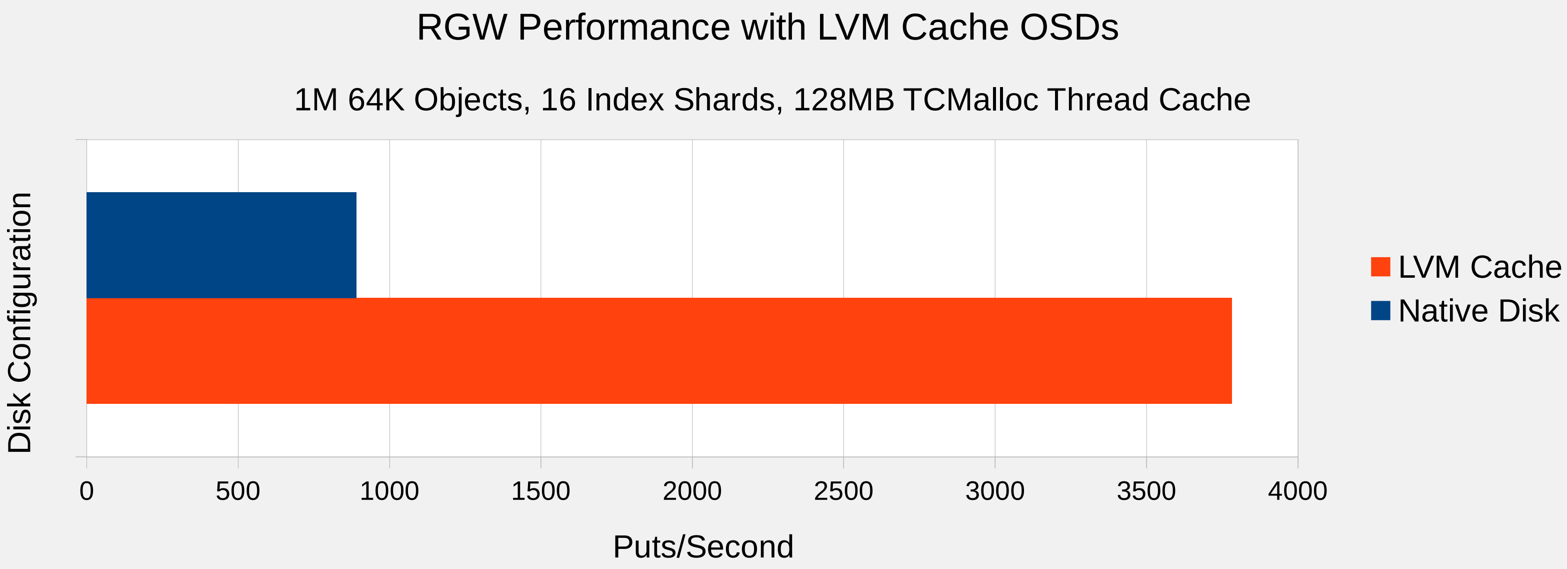
What about filestore?

- Put journals for rgw.buckets pool on SSD to avoid journal penalty
- Put rgw.buckets.index pools on SSD backed OSDs
- More OSDs for rgw.buckets.index means more PGs, higher memory usage, and potentially lower distribution quality.
- Are there alternatives?

RGW BUCKET INDEX OVERHEAD

Are there alternatives? Potentially yes!

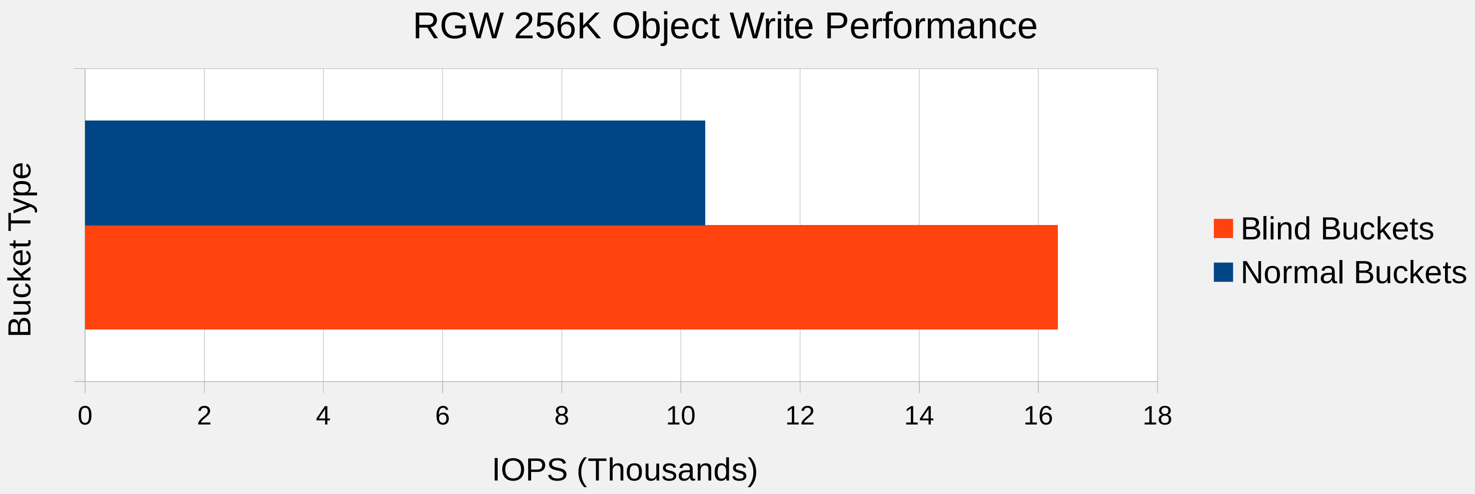
Ben England from Red Hat's Performance Team is testing RGW with **LVM Cache**. Initial (100% cached) performance looks promising. Will it scale though?



RGW BUCKET INDEX OVERHEAD

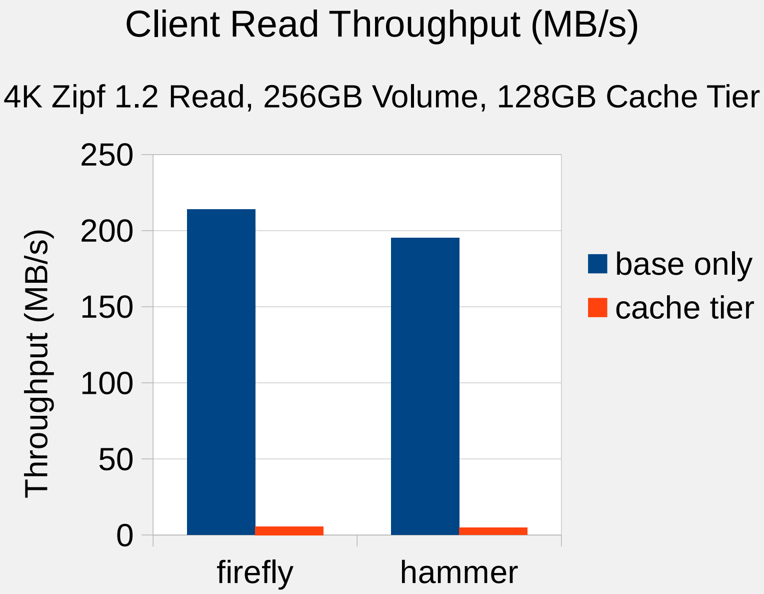
What if you don't need bucket indices?

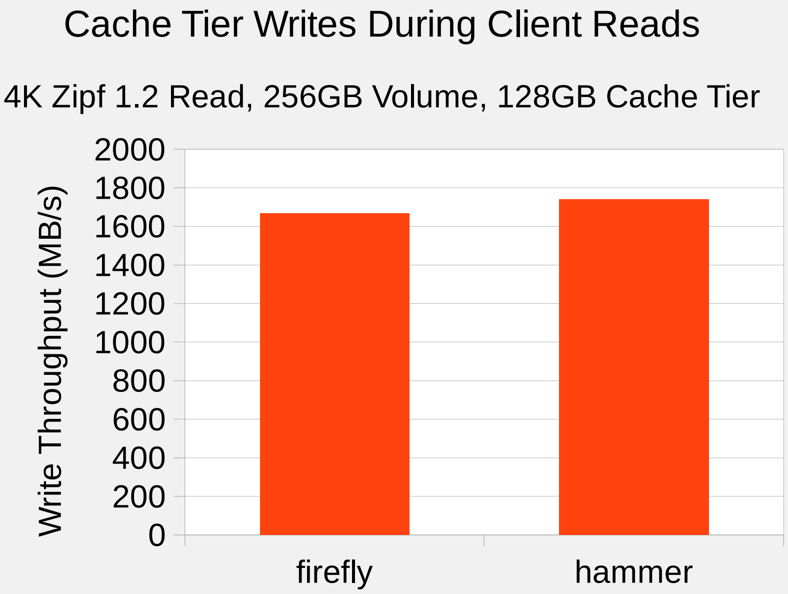
The customer we tested with didn't need Ceph to keep track of bucket contents, so for Jewel we introduce the concept of ***Blind Buckets*** that do not maintain bucket indices. For this customer the overall performance improvement was near 60%.



CACHE TIERING

The original cache tiering implementation in firefly was very slow when





CACHE TIERING

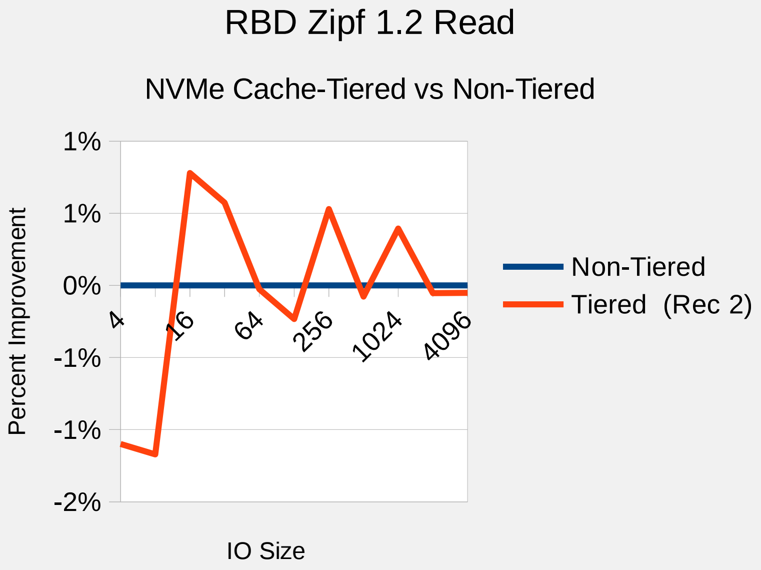
There have been many improvements since then...

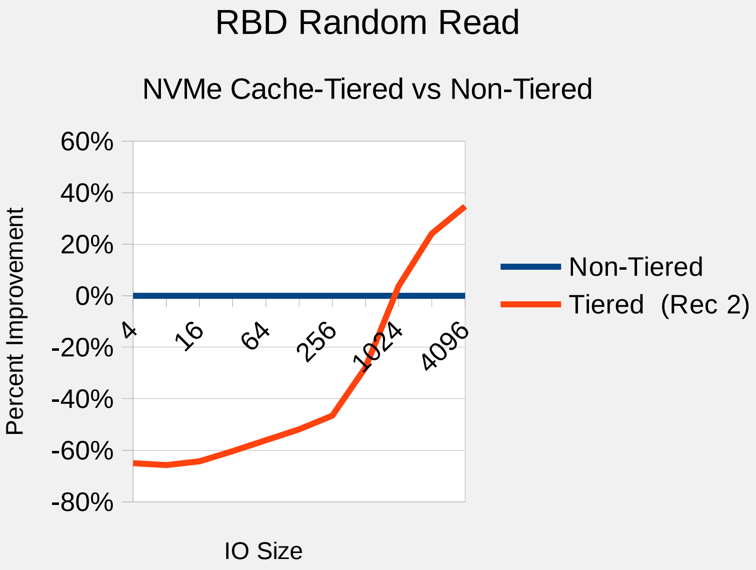
- Memory Allocator tuning helps SSD tier in general
- Read proxy support added in Hammer
- Write proxy support added in Infernalis
- Recency fixed: https://github.com/ceph/ceph/pull/6702
- Other misc improvements.
- Is it enough?

CACHE TIERING

Is it enough?

Zipf 1.2 distribution reads performing very similar between base only and tiered but random reads are still slow at small IO sizes.

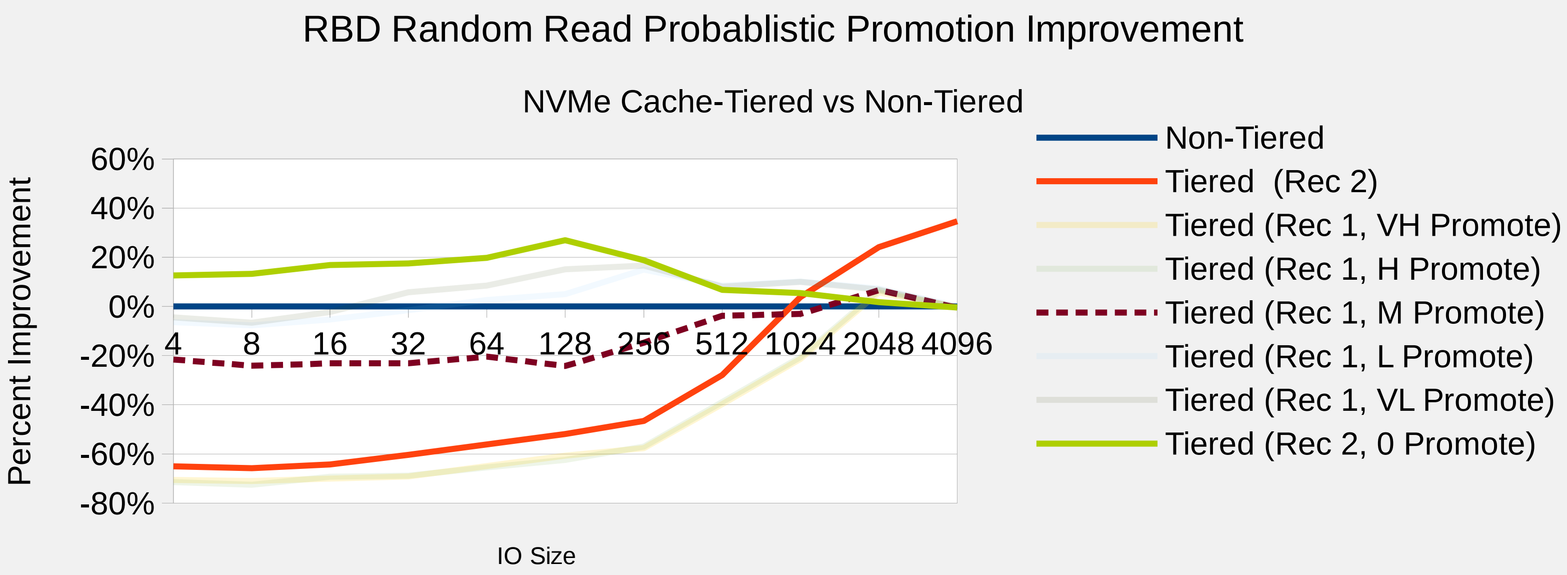




CACHE TIERING

Limit promotions even more with object and throughput throttling.

Performance improves dramatically and in this case even beats the base tier when promotions are throttled very aggressively.



RBD Random Read Probablistic Promotion Improvement

NVMe Cache-Tiered vs Non-Tiered

CACHE TIERING

Conclusions

- Performance very dependent on promotion and eviction rates.
- Limiting promotion can improve performance, but are we making the cache tier less adaptive to changing hot/cold distributions?
- Will need a lot more testing and user feedback to see if our default promotion throttles make sense!

redhat

THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHatNews

youtube.com/user/RedHatVideos