Ceph and Flash Storage
Brian O’Krafka
Sandisk Engineering Fellow
April 20, 2016
Forward Looking Statements

During our meeting today, we may make forward-looking statements.

Any statement that refers to expectations, projections or other characterizations of future events or circumstances is a forward-looking statement, including those relating to market position, market growth, product sales, industry trends, supply chain, future memory technology, production capacity, production costs, technology transitions, construction schedules, production starts, and future products. This presentation contains information from third parties, which reflect their projections as of the date of issuance. Actual results may differ materially from those expressed in these forward-looking statements due to factors detailed under the caption “Risk Factors” and elsewhere in the documents we file from time to time with the SEC, including our annual and quarterly reports. We undertake no obligation to update these forward-looking statements, which speak only as of the date hereof or the date of issuance by a third party, as the case may be.
Overview

- Credits
- Why is Sandisk interested in Ceph?
- Optimizing Ceph for flash
  - Read and write path optimizations
  - Ceph storage engine optimizations
- Key-value storage engines
  - Taxonomy
  - RocksDB, ZetaScale
- Ceph/RocksDB vs. Ceph/ZetaScale
- Project Outlook
Credits

- The material in this presentation is the work of a large team of contributors:
  - Allen Samuels: father of EMS Ceph strategy
  - Somnath Roy, Sushma Gurram, Chaitanya Huilgol: early Ceph read and write path optimizations
  - Mana Krishnan and ZetaScale team: original filestore/ZS integration, current Bluestore/ZS work
  - EMS Ceph QA team: QA and performance testing
  - EMS product management, marketing and sales
Why is Sandisk interested in Ceph?
Ceph: Open-Source Unified FOB Storage Software

>1000 Node Deployments

Ceph is gaining momentum!
InfiniFlash IF500 – The fastest, most affordable all flash storage for OpenStack/CEPH deployments

Cost-effective, high performance, scalable all-flash storage for cloud and big data workloads

- Ultra-dense All-Flash Appliance
  - 512TB in 3U
  - Best in class $/IOPS/TB
- Scale-out software for massive capacity
  - Unified Content: Block, Object
  - Flash optimized software with programmable interfaces (SDK)
- Enterprise-Class storage features
  - snapshots, replication, thin provisioning
  - 10X the performance of HDD based deployments
- Targeted for OpenStack/Ceph deployments
- Starting at <$1/GB
InfiniFlash 12G Raw Performance

**Environment:**
- Raw IF150 IO perf is collected on full pop A8 topology setup
- 64 BSSDs mapped to 8 Hosts using slot mapping
- Pre-condition drives by 256k random write for 30hrs after drive format
- Trigger Performance using vdbench tool.

**8 Node Topology**

**Performance**

<table>
<thead>
<tr>
<th>IO Profile</th>
<th>Read IOPs</th>
<th>Read BW</th>
</tr>
</thead>
<tbody>
<tr>
<td>4k RR</td>
<td>2448k</td>
<td>9.8GB/s</td>
</tr>
<tr>
<td>64k RR</td>
<td>213k</td>
<td>13.6GB/s</td>
</tr>
<tr>
<td>256k RR</td>
<td>50k</td>
<td>12.8GB/s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IO Profile</th>
<th>Write IOPs</th>
<th>Write BW</th>
</tr>
</thead>
<tbody>
<tr>
<td>4k RW</td>
<td>334k</td>
<td>1.3GB/s</td>
</tr>
<tr>
<td>64k RW</td>
<td>97k</td>
<td>6.2GB/s</td>
</tr>
<tr>
<td>256k RW</td>
<td>24k</td>
<td>6.1GB/s</td>
</tr>
</tbody>
</table>
Optimizing Ceph for Flash
Flash Optimization

- Flash-optimized applications:
  - Exploit the high capacity, low latency, persistence and high throughput of flash memory
  - Have extensive parallelism to enable many concurrent flash accesses for high throughput
  - Use DRAM as a cache for hot data

- Applications realize limited benefits from flash without additional optimization

- Optimizing Ceph for flash:
  - read path optimization
  - write path optimization
  - key-value storage engine optimization
Ceph Read Path Optimizations (In Hammer)

- Messenger layer:
  - removed dispatcher and introduced a “fast path” mechanism
  - finer-grained locking in message transmit path
  - made buffering more efficient

- OSD request processing:
  - added sharded worker thread pool: requests sharded by placement group identifier
  - configurable number of shards and worker threads per shard
  - optimized OpTracking path by sharding queue and removing redundant locks

- FileStore improvements:
  - sharded LRU cache to reduce lock contention in LRU file descriptor cache
  - CollectionIndex object (per-PB) was created for every I/O request: added cache for these since PG info does not change often
  - optimized object-name-to XFS filename mapping function
  - removed redundant snapshot checks in parent read processing path

- Issues with TCmalloc
  - I/O from different clients assigned to different threads: memory movement among thread caches, increasing alloc/free latency
  - Use JEmalloc or increase TCmalloc thread cache size

- Various Client Optimizations (turn off Nagle’s algorithm, do not ignore TCP_NODELAY)
Ceph RBD Read Performance with Read Optimizations

Throughput (IOPs) 8kB Block, 1 RBD/Client 32 OSDs per Node Latency (ms)

Queue Depth, %Reads

Queue Depth, %Reads

SanDisk 1x 512TB IF100; 2x Ceph Nodes: Dell R620, ea: 2x E5-2680, 12c, 2.8GHz, 64GB DRAM; 4x RBD Clients: Dell R620, ea: 2x E5-2680, 10c, 2.8GHz, 32GB DRAM; 40G ethernet
Ceph Write Path Optimizations (In Jewel)

- New I/O throttling scheme for writes:
  - based on journal usage and outstanding IO
  - joint effort RedHat/SanDisk

- big syncfs problematic with SSDs
  - replaced with O_DSYNC write for each I/O
  - higher, more stable throughput.

- Efficient transaction deletion with the help of move constructor semantics

- Reduced CPU usage throughout the code base
Ceph RBD Write Performance with Write Optimizations (Dev Data)

- 3xIF100, 16 SSD’s each
- 6xHead Nodes, each: 2xXeon E5-2695 v2 @ 2.4GHz, 48 threads, 64GB DRAM
- 1 or 2 clients, each: 1x Xeon E5-2680 v2 @2.8GHz, 20 threads, 64GB DRAM
- Stock: Hammer, Opt: Jewel
- 2 pools and two images (one per pool) of 2TB each
- Replication = 2
- Each image preconditioned with 256K sequential writes
- FIO benchmark, 10 num_jobs with 128QD each
Storage Engines
OBJECT STORAGE DAEMONS (OSDS)
Problems with FileStore

- **Transactions**
  - All OSD writes are transactions (typical: write file data + update xattr(s) + leveldb insert)
  - No good transaction support in file systems (btrfs, xfs, ext4)
  - Forced to use write-ahead journals: cut disk throughput in half

- **Enumeration**
  - Must be able to enumerate objects by 32-bit Ceph hash (scrubbing, data rebalancing, client API)
  - POSIX readdir does not work
  - Solution: directory tree using hash-value prefix; ~100 files/dir, read entire directories and sort in memory

- **BlueStore solves these problems**
  - BlueStore = NewStore over Block (NewStore = Rocksdb + object files)
  - Rocksdb supports transactions and enumeration by hash
  - Replacing file system with simple block layer avoids duplicate journaling when using rocksdb on a file system
BLUESTORE DESIGN

- rockbdb
  - object metadata (onode) in rockbdb
  - write-ahead log (small writes/overwrites)
  - ceph key/value “omap” data
  - allocator metadata (free extent list)
- block device
  - object data
- pluggable allocator
- rockbdb shares block device(s)
  - BlueRocksEnv is rockbdb::Env
  - BlueFS is super-simple C++ “file system”
- 2x faster on HDD, more on SSD
Ceph Block Write Path

- Typical Ceph block write operation: several small xattr writes plus data write plus ceph log write(s)
- Large data values written first directly to block device; old copy of data is left in place (copy-on-write)
- Attributes, onode writes then sent to RocksDB (write-behind)
- Multiple RocksDB writes are enclosed in RocksDB transaction for crash-safety
- Small data writes get special handling to deal with crash-safety of read-modify-write operations on data block device:
  - small data writes are first logged via RocksDB (in write-ahead log [WAL] entries)
  - after RocksDB txn commit, WAL entries are applied to data block device
- fsync applied after large data writes, then after RocksDB commit
Key-Value Store Operations

- Key-value instance:
  - open, close
  - get/set parameters, stats

- Containers:
  - open, close, delete, snapshot
  - enumerate containers
  - enumerate objects

- Objects:
  - read, write, delete
  - multi-read/write/delete
  - query

- Transactions:
  - start, commit

Settings:

- DRAM cache size
- flash device or file(s)
- flash size
- per-container properties (extensible):
  - quota (can be unlimited)
  - durability:
    - periodic
    - software crash-safe
    - hardware crash-safe
## Alternative Algorithms for Key-Value Stores

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Point Read-Amp</th>
<th>Range Read-Amp</th>
<th>Write-Amp</th>
<th>Space-Amp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>B-tree: update-in-place (UIP)</strong></td>
<td>1</td>
<td>1-2</td>
<td>node/data * SSD-WA</td>
<td>1.5-2</td>
</tr>
<tr>
<td><strong>B-tree: copy-on-write-random (COW-R)</strong></td>
<td>1</td>
<td>1-2</td>
<td>node/data * SSD-WA</td>
<td>1.5-2</td>
</tr>
<tr>
<td><strong>B-tree: copy-on-write-sequential (COW-S)</strong></td>
<td>1</td>
<td>1-2</td>
<td>node/data * SW-WA</td>
<td>1.5-2</td>
</tr>
<tr>
<td><strong>LSM: leveled compaction (N levels)</strong></td>
<td>1 + N*bloom</td>
<td>N</td>
<td>10 per level</td>
<td>1.1</td>
</tr>
<tr>
<td><strong>LSM: universal compaction (N files)</strong></td>
<td>1 + N*bloom</td>
<td>N</td>
<td>typically &lt; 10</td>
<td>&gt;2</td>
</tr>
<tr>
<td>log-only</td>
<td>1</td>
<td>N</td>
<td>1/(1-%live)</td>
<td>1/,%live</td>
</tr>
<tr>
<td>memtable-L1</td>
<td>1</td>
<td>1</td>
<td>database/mem</td>
<td>1</td>
</tr>
<tr>
<td>memtable+L0+L1 (N files)</td>
<td>1 + N*bloom</td>
<td>N</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>tokudb</td>
<td>1</td>
<td>2</td>
<td>10 per level</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Rocksdb Write Path

Write

Memory

Active Memtable

Queue for Flushing

Inactive Memtable

Inactive Memtable

Persist as SSTable

Disk

Log

Persist as SSTable

Disk

SSTable

Compaction

ref: Siying Dong, *Embedded Key-Value Store for Flash and Faster Storage*, Flash Memory Summit 2014
Rocksdb: Leveled Compaction

- Read Amp: $1 + N \times \text{Bloom}$
- Write Amp: $10 \times N$
- Space Amp: 1.1 (theoretical)

- No overlapping keys in L1, L2, ...
- Compaction of $L_i$ with $L_{i+1}$ triggered by $L_i$ size, number of files, etc.

ref: Siying Dong, *Embedded Key-Value Store for Flash and Faster Storage*, Flash Memory Summit 2014
Rocksdb Universal Compaction

- Read Amp: $1 + N \cdot \text{Bloom}$
- Write Amp: $\leq N$
- Space Amp: 2
  (Need double space for compaction)

- Each level covers disjoint chronological period of time
- Levels at higher levels are older
- Compact $L_i$ with $L_{i+1}$ based on file sizes, number of files, current space amplification, etc.
## Rocksdb: Leveled vs Universal Compaction

For 1TB Database with 1GB Flush Size

<table>
<thead>
<tr>
<th></th>
<th>Point Read-Amp</th>
<th>Range Read-Amp</th>
<th>Write-Amp</th>
<th>Space-Amp</th>
<th>Double Space Issue?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSM: leveled compaction (5 levels)</td>
<td>1 + N*bloom</td>
<td>5</td>
<td>40</td>
<td>1.1</td>
<td>No</td>
</tr>
<tr>
<td>LSM: universal compaction (11 regions)</td>
<td>1 + N*bloom</td>
<td>11</td>
<td>&lt;=11</td>
<td>2</td>
<td>Yes</td>
</tr>
</tbody>
</table>

ref: Siying Dong, *Embedded Key-Value Store for Flash and Faster Storage*, Flash Memory Summit 2014
ZetaScale Internals

High-level Data flow

① Objects are written to Btree & L1cache
  - Non-leaf nodes hold indexes
  - Leaf nodes hold the objects
  - Larger objects are stored in overflow nodes
② Btree uses core object interface to write/persist btree nodes as key-value objects
③ Slab allocator allocates a slab for the object from flash
④ A mapping entry(key to slab) is created in the Hash table or corresponding bit is set in the slab allocator if the object is large
⑤ Object is written to flash
⑥ A log entry which holds mapping info for the object is created
⑦ Application request is acknowledged
ZetaScale B-Tree

Key Features

- Implemented on top of core container, object, transaction API
  - flash optimized using these mechanisms
  - B-tree nodes automatically cached via object DRAM cache
  - write-through L1 “pinning” cache
- Highly concurrent access
- Efficient index queries
- Point-in-time snapshots for full and incremental backups
- Crash safe
- Efficient packing of small objects
- Highly configurable
  - Node size
  - Sort compare functions
  - Unlimited data size

- Each Btree node is stored as object with unique logical ID as key
- Data kept in the leaves
- Large data items overflow to linked list of nodes
RocksDB vs. ZetaScale
Test Configuration and Setup

### ZetaScale configuration
- ZS Version: 3.0-2757.223
- L1 Cache: 48G
- Btree Containers
- Durability (Hardware Crash Safe)
- ZS Logs on NVRAM backed HDD

### RocksDB configuration
- RocksDB Version: 3.12.0
- Compaction Type:
  - Universal compaction (Write AMP friendly & small dataset)
  - Leveled (recommended for performance & larger dataset)
- Durability: (Hardware Crash Safe)
- Write-ahead Logs on NVRAM backed HDD
- Memtable size: 128M
- Number of compaction Threads: 32
- RocksDB uses system buffercache (~60G) as cache.
- Other Configurations:

```
rocksdb.block_cache=1073741824
rocksdb.db=/schooner/data/
rocksdb.max_background_compactions=32
rocksdb.max_background_flushes=1
rocksdb.level0_file_num_compaction_trigger=4
rocksdb.level0_slowdown_writes_trigger=8
rocksdb.level0_stop_writes_trigger=16
rocksdb.compaction_style=0
rocksdb.max_bytes_for_level_base=536870912
rocksdb.max_write_buffer_number=2
rocksdb.write_buffer_size=134217728
rocksdb.target_file_size_base=53687091
rocksdb.stats_dump_period_sec=120
rocksdb.max_open_files=-1
rocksdb.max_grandparent_overlap_factor=10
```
### Performance, WriteAmp Summary (Universal Compaction, 128G Dataset)

<table>
<thead>
<tr>
<th>Workload Read/Write</th>
<th>object size</th>
<th>Throughput MB/sec - 100% write</th>
<th>Throughput MB/sec - 95% Read, 5% Write</th>
<th>Throughput MB/sec - 70% Read, 30% Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1K</td>
<td>16K</td>
<td>64K</td>
</tr>
<tr>
<td>0/100</td>
<td>1K</td>
<td>39.274</td>
<td>3.494</td>
<td>0.942</td>
</tr>
<tr>
<td></td>
<td>16K</td>
<td>34.262</td>
<td>10.295</td>
<td>5.491</td>
</tr>
<tr>
<td></td>
<td>64K</td>
<td>13.35</td>
<td>11.88</td>
<td>12.27</td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>12.02</td>
<td>4.53</td>
<td>2.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Performance improvement by ZetaScale</td>
<td>RocksDB WA Reduction by ZetaScale</td>
</tr>
<tr>
<td></td>
<td>95/5</td>
<td>1K</td>
<td>143.573</td>
<td>96.396</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16K</td>
<td>24.167</td>
<td>36.828</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64K</td>
<td>8.788</td>
<td>19.368</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1M</td>
<td>0.557</td>
<td>1.791</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>26.26</td>
<td>9.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15.25</td>
<td>4.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14.87</td>
<td>2.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15.04</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14.48</td>
<td>9.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16.19</td>
<td>4.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14.77</td>
<td>2.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15.71</td>
<td>1.07</td>
</tr>
<tr>
<td>70/30</td>
<td>1K</td>
<td>79.942</td>
<td>24.596</td>
<td>473%</td>
</tr>
<tr>
<td></td>
<td>16K</td>
<td>8.061</td>
<td>11.665</td>
<td>1315%</td>
</tr>
<tr>
<td></td>
<td>64K</td>
<td>2.037</td>
<td>11.665</td>
<td>1315%</td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>0.114</td>
<td>1.608</td>
<td>1315%</td>
</tr>
</tbody>
</table>

**Write Amplification**

- **Object Size**: 1K, 16K, 64K, 1M
- **Write Amplification** (WA) Reduction by ZetaScale:
  - 1K: 33%
  - 16K: 52%
  - 64K: 120%
  - 1M: 1315%

**Workload Read/Write**

- **ZetaScale**
- **RocksDB**
- **Performance improvement by ZetaScale**
- **WA Reduction by ZetaScale**

---

**SanDisk**
Performance, WriteAmp Summary (Leveled Compaction, 128G Dataset)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Read/Write</th>
<th>object size</th>
<th>RocksDB KTPS</th>
<th>ZetaScale KTPS</th>
<th>Performance improvement by ZetaScale</th>
<th>RocksDB WA</th>
<th>ZetaScale WA</th>
<th>WA Reduction by ZetaScale</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/100</td>
<td>1K</td>
<td>50.970</td>
<td>34.262</td>
<td>-33%</td>
<td>18.16</td>
<td>12.02</td>
<td>34%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16K</td>
<td>4.266</td>
<td>10.295</td>
<td>141%</td>
<td>15.38</td>
<td>4.53</td>
<td>71%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64K</td>
<td>1.053</td>
<td>5.491</td>
<td>421%</td>
<td>16.17</td>
<td>2.14</td>
<td>87%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>0.079</td>
<td>0.697</td>
<td>780%</td>
<td>13.62</td>
<td>1.08</td>
<td>92%</td>
<td></td>
</tr>
<tr>
<td>95/5</td>
<td>1K</td>
<td>186.798</td>
<td>96.396</td>
<td>-48%</td>
<td>22.62</td>
<td>9.05</td>
<td>60%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16K</td>
<td>48.632</td>
<td>36.828</td>
<td>-24%</td>
<td>19.42</td>
<td>4.52</td>
<td>77%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64K</td>
<td>13.985</td>
<td>19.368</td>
<td>38%</td>
<td>17.45</td>
<td>2.13</td>
<td>88%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>0.888</td>
<td>1.791</td>
<td>102%</td>
<td>16.45</td>
<td>1.07</td>
<td>93%</td>
<td></td>
</tr>
<tr>
<td>70/30</td>
<td>1K</td>
<td>100.368</td>
<td>61.522</td>
<td>-39%</td>
<td>20.91</td>
<td>9.05</td>
<td>57%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16K</td>
<td>11.726</td>
<td>24.596</td>
<td>110%</td>
<td>17.36</td>
<td>4.52</td>
<td>74%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64K</td>
<td>3.045</td>
<td>11.665</td>
<td>283%</td>
<td>17.02</td>
<td>2.13</td>
<td>87%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>0.203</td>
<td>1.608</td>
<td>694%</td>
<td>15.92</td>
<td>1.07</td>
<td>93%</td>
<td></td>
</tr>
</tbody>
</table>

Throughput - 100% write

Throughput - 95% Read, 5% Write

Throughput - 70% Read, 30% Write

Write Amplification Object Size

Throughput MB/sec Object Size

ZetaScale RocksDB

SanDisk
Performance, WriteAmp Summary (Leveled Compaction, 0.75TB dataset)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Read/Write</th>
<th>object size</th>
<th>RocksDB KTPS</th>
<th>ZetaScale KTPS</th>
<th>Performance improvement by ZetaScale</th>
<th>ZetaScale WA</th>
<th>WA Reduction by ZetaScale</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/100</td>
<td></td>
<td>64000</td>
<td>0.677</td>
<td>9.761</td>
<td>1342%</td>
<td>21.47</td>
<td>1.13 95%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1M</td>
<td>0.046</td>
<td>0.758</td>
<td>1546%</td>
<td>19.82</td>
<td>1.07 95%</td>
</tr>
<tr>
<td>95/5</td>
<td></td>
<td>64000</td>
<td>8.358</td>
<td>34.967</td>
<td>318%</td>
<td>26.71</td>
<td>1.13 96%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1M</td>
<td>0.556</td>
<td>1.777</td>
<td>219%</td>
<td>23.83</td>
<td>1.07 95%</td>
</tr>
<tr>
<td>70/30</td>
<td></td>
<td>64000</td>
<td>1.796</td>
<td>23.982</td>
<td>1235%</td>
<td>25.50</td>
<td>1.13 96%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1M</td>
<td>0.125</td>
<td>1.381</td>
<td>1006%</td>
<td>22.54</td>
<td>1.07 95%</td>
</tr>
</tbody>
</table>

Write Amplification

Throughput - 100% write

Throughput - 95% read, 5% write

Throughput - 70% read, 30% write

<table>
<thead>
<tr>
<th>Workload</th>
<th>Read/Write</th>
<th>object size</th>
<th>MB/sec</th>
<th>MB/sec</th>
<th>MB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/100</td>
<td></td>
<td>64000</td>
<td>600</td>
<td>400</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1M</td>
<td>800</td>
<td>600</td>
<td>400</td>
</tr>
<tr>
<td>95/5</td>
<td></td>
<td>64000</td>
<td>2500</td>
<td>1500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1M</td>
<td>3000</td>
<td>2000</td>
<td>1500</td>
</tr>
<tr>
<td>70/30</td>
<td></td>
<td>64000</td>
<td>1600</td>
<td>1200</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1M</td>
<td>2000</td>
<td>1500</td>
<td>1000</td>
</tr>
</tbody>
</table>

SanDisk
Throughput Stability: 64K Object size (Leveled Compaction, 128G Dataset)

100% Write

70% Read, 30% Write
Observation:
- ZetaScale throughput is stable
- RocksDB throughput is choppy and unstable in both write and read intensive workloads. Choppiness gets worse with large dataset.
Ceph/RocksDB vs. Ceph/ZetaScale
(Preliminary Results)
Ceph RBD Performance: RocksDB vs ZetaScale

- Single Ceph OSD over 8TB Infiniflash SSD + single LIBRBD block client connected to OSD over 10G network

- Server:
  - 2x Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz – 20 cores/40 threads total
  - 1x Infiniflash SSD – 8TB
  - 8GB memory
  - 10G ethernet
  - Ceph revisions: 88a183b (Jewel) / 99c6f30 (master)

- Client:
  - 2x Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz – 12 cores/24 threads total
  - 128GB memory
  - 10G Ethernet

- Workload: fio-2.1.3
  - 32 threads / 64 iodepth (reducing either value yields lower performance)
  - 6TB of data
  - Load with default (4MB) rbd object size
  - 4kB block fio-rbd writes/reads
  - 6 hours warmup (100% write)
  - 15 minute measurement runs: 100%write, 70/30, 99/1 (due to omap trim bug)

- Ceph options:
  - rbd_cache = false
  - rbd_non_blocking_aio = false
  - rbd_op_threads = 1
  - cephx_sign_messages = false
  - cephx_require_signatures = false
  - osd_op_num_threads_per_shard = 8
  - osd_op_num_shards = 32
  - osd pool default size = 1
  - osd_op_threads = 32
  - osd_disk_threads = 32
  - osd objectstore = bluestore
  - bluestore_kvbackend=rocksdb
  - bluestore_bluefs=true
  - bluestore_min_alloc_size=4096
  - bluestore_block_size=7696581394432
  - bluestore_block_path=/dev/sdb

- Explicit ceph rocksdb options:
  - set rocksdb option compression = kNoCompression
  - set rocksdb option db_log_dir = /tmp/rocksdb_stats
  - set rocksdb option max_write_buffer_number = 16
  - set rocksdb option min_write_buffer_number_to_merge = 3
  - set rocksdb option recycle_log_file_num = 16
Ceph RBD Performance: RocksDB vs ZetaScale

### kIOPs (4kB Block)

<table>
<thead>
<tr>
<th></th>
<th>100% write</th>
<th>70% Read</th>
<th>99% Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>RocksDB</td>
<td>1.1</td>
<td>2.6</td>
<td>14.9</td>
</tr>
<tr>
<td>ZS</td>
<td>1.1</td>
<td>3.5</td>
<td>35.1</td>
</tr>
</tbody>
</table>

### Write Amplification (4kB Block)

<table>
<thead>
<tr>
<th></th>
<th>100% write</th>
<th>70% Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>RocksDB</td>
<td>8.9</td>
<td>8.6</td>
</tr>
<tr>
<td>ZS</td>
<td>10.9</td>
<td>10.9</td>
</tr>
</tbody>
</table>

### CPU ms Per Client IOP (4kB Block)

<table>
<thead>
<tr>
<th></th>
<th>100% Write</th>
<th>70% Read</th>
<th>99% Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>RocksDB</td>
<td>1.85</td>
<td>0.83</td>
<td>0.35</td>
</tr>
<tr>
<td>ZS</td>
<td>0.70</td>
<td>0.34</td>
<td>0.24</td>
</tr>
</tbody>
</table>
Ceph Throughput Stability: 4K Block (6TB Dataset)

- **100% Write**
- **70% Read, 30% Write**
Project Outlook
Apply Some Optimizations from Old Integration Strategy (ZS/FileStore)

- Optimize mapping of object store write sequences to ZS interface:
  - Group writes of adjacent objects into a single bulk write
  - Use a logging container for Ceph log writes (coalesce multiple small log writes in NVRAM to minimize write amplification)

- Shim responsible for:
  - breaking large objects into 64k chunks (post-compression)
  - Compression/decompression
  - Mapping partial updates and appends into operations on 64k chunks

- Typical Ceph block write operation:
  - Maps to several small xattr writes plus data write plus ceph log write(s)
  - Xattr and data writes are adjacent in B-tree and combined in a single ZS multiput operation
  - For 64kB or smaller block write:
    - 1 8kB B-tree leaf node write
    - 1 64kB data block write
    - Ceph log writes accumulate in NVRAM via ZS log container
      - Write amp = (64kB + 8kB)/64kB = 1.125
      - Write amp will be lower for larger block sizes
### Alternative Algorithms for Key-Value Stores

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Point Read-Amp</th>
<th>Range Read-Amp</th>
<th>Write-Amp</th>
<th>Space-Amp</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-tree: update-in-place (UIP)</td>
<td>1</td>
<td>1-2</td>
<td>node/data * SSD-WA</td>
<td>1.5-2</td>
</tr>
<tr>
<td><strong>B-tree: copy-on-write-random (COW-R)</strong></td>
<td>1</td>
<td>1-2</td>
<td>node/data * SSD-WA</td>
<td>1.5-2</td>
</tr>
<tr>
<td>B-tree: copy-on-write-sequential (COW-S)</td>
<td>1</td>
<td>1-2</td>
<td>node/data * SW-WA</td>
<td>1.5-2</td>
</tr>
<tr>
<td>LSM: leveled compaction (N levels)</td>
<td>$1 + N*bloom$</td>
<td>$N$</td>
<td>10 per level</td>
<td>1.1</td>
</tr>
<tr>
<td>LSM: universal compaction (N files)</td>
<td>$1 + N*bloom$</td>
<td>$N$</td>
<td>typically &lt; 10</td>
<td>&gt;2</td>
</tr>
<tr>
<td>log-only</td>
<td>1</td>
<td>$N$</td>
<td>$1/(1-%live)$</td>
<td>$1/%live$</td>
</tr>
<tr>
<td>memtable-L1</td>
<td>1</td>
<td>1</td>
<td>database/mem</td>
<td>1</td>
</tr>
<tr>
<td>memtable+L0+L1 (N files)</td>
<td>$1 + N*bloom$</td>
<td>$N$</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>tokudb</td>
<td>1</td>
<td>2</td>
<td>10 per level</td>
<td>1.1</td>
</tr>
</tbody>
</table>

**Ref:** Mark Callaghan, *MySQL versus something else: Evaluating alternative databases*, Highload++ 2013
Benefits of Write Serialization

- 8T Infiniflash SSD ("Ice-Chip").
- 4k Random Writes.
- Test: FIO, 256 Threads
- System: DELL R720 32 cores and 128GB DRAM.
Conclusion

- Sandisk is committed to Ceph as a strategic open source solution for flash storage

- Ceph must be flash-optimized to extract the full value of flash storage
  - This is particularly significant for Sandisk’s Infiniflash products with 100’s TB of capacity and millions of IOPs

- Sandisk is investing in open source enhancements for Ceph’s read path, write path and back-end storage engine
  - Read path optimizations have yielded up to 10x improvement in block read IOPs
  - Write path optimizations yield a 2x improvement in 4k IOPs
  - Sandisk ZetaScale is a promising open source storage engine when large amounts of flash is managed with relatively little host DRAM
Thank You!
brian.okrafka@sandisk.com