Novel algorithms of the ZFS storage system

Matt Ahrens
Principal Engineer, Delphix
ZFS co-creator
Brown CS 2001
Established in 1764, Brown is a liberal arts University of some 5,500 undergraduates located in Providence, Rhode Island. A member of the Ivy League, Brown is known for its flexible curriculum and opportunities for independent study and collaborative work with faculty at all levels. Brown has a School of Medicine (300 students) and Graduate School (1,300 students).
Talk overview

● History
● Overview of the ZFS storage system
● How ZFS snapshots work
● ZFS on-disk structures
● How ZFS space allocation works
● How ZFS RAID-Z works
● Future work
ZFS History

- 2001: development starts at Sun with 2 engineers
- 2005: ZFS source code released
- 2008: ZFS released in FreeBSD 7.0
- 2010: Oracle stops contributing to source code for ZFS
- 2010: illumos is founded as the truly open successor to OpenSolaris
- 2013: ZFS on (native) Linux GA
- 2013: Open-source ZFS bands together to form OpenZFS
- 2014: OpenZFS for Mac OS X launch
Talk overview

- History
- **Overview of the ZFS storage system**
  - How ZFS snapshots work
  - ZFS on-disk structures
  - How ZFS space allocation works
  - How ZFS RAID-Z works
- Future work
Overview of ZFS

- **Pooled storage**
  - Functionality of filesystem + volume manager in one
  - Filesystems allocate and free space from pool

- **Transactional object model**
  - Always consistent on disk (no FSCK, ever)
  - Universal - file, block, NFS, SMB, iSCSI, FC, …

- **End-to-end data integrity**
  - Detect & correct silent data corruption

- **Simple administration**
  - Filesystem is the administrative control point
  - Inheritable properties
  - Scalable data structures
```bash
zpool create tank raidz2 d1 d2 d3 d4 d5 d6
zfs create tank/home
zfs set sharenfs=on tank/home
zfs create tank/home/mahrens
zfs set reservation=10T tank/home/mahrens
zfs set compression=gzip tank/home/dan
zpool add tank raidz2 d7 d8 d9 d10 d11 d12
zfs create -o recordsize=8k tank/DBs
zfs snapshot -r tank/DBs@test
date
zfs clone tank/DBs/prod@test tank/DBs/test
date
```
Copy-On-Write Transaction Groups (TXG’s)

1. Initial block tree

2. COW some blocks

3. COW indirect blocks

4. Rewrite uberblock (atomic)
Bonus: Constant-Time Snapshots

- The easy part: at end of TX group, don't free COWed blocks
Talk overview

- History
- Overview of the ZFS storage system
- **How ZFS snapshots work**
- ZFS on-disk structures
- How ZFS space allocation works
- How ZFS RAID-Z works
- Future work
ZFS Snapshots

● How to create snapshot?
  ○ Save the root block

● When block is removed, can we free it?
  ○ Use BP’s birth time
  ○ If birth > prevsnap
    ■ Free it

● When delete snapshot, what to free?
  ○ Find unique blocks - Tricky!
Trickiness will be worth it!

**Per-Snapshot Bitmaps**
- Block allocation bitmap for every snapshot
  - O(N) per-snapshot space overhead
  - Limits number of snapshots
- O(N) create, O(N) delete, O(N) incremental
  - Snapshot bitmap comparison is O(N)
  - Generates unstructured block delta
  - Requires some prior snapshot to exist

**ZFS Birth Times**
- Each block pointer contains child's birth time
  - O(1) per-snapshot space overhead
  - Unlimited snapshots
- O(1) create, O(Δ) delete, O(Δ) incremental
  - Birth-time-pruned tree walk is O(Δ)
  - Generates semantically rich object delta
  - Can generate delta since any point in time
Snapshot Deletion

- Free unique blocks (ref’d only by this snap)
- Optimal algo: \(O(\# \text{ blocks to free})\)
  - And \(\# \text{ blocks to read from disk } \ll \# \text{ blocks to free}\)
- Block lifetimes are contiguous
  - AKA “there is no afterlife”
  - Unique = not ref’d by prev or next (ignore others)
Snapshot Deletion

- Traverse tree of blocks
- Birth time <= prev snap?
  - Ref’d by prev snap; do not free.
  - Do not examine children; they are also <= prev

Older snap #19
Prev snap #25
Deleting snap #37

37 19
37 25
19 15
Snapshot Deletion

- Traverse tree of blocks
- Birth time $\leq$ prev snap?
  - Ref’d by prev snap; do not free.
  - Do not examine children; they are also $\leq$ prev
- Find BP of same file/offset in next snap
  - If same, ref’d by next snap; do not free.
- $O(\# \text{ blocks written since prev snap})$
- How many blocks to read?
  - Could be $2x \# \text{ blocks written since prev snap}$
Snapshot Deletion (ﷺ)

- Read Up to 2x # blocks written since prev snap
- Maybe you read a million blocks and free nothing
  - (next snap is identical to this one)
- Maybe you have to read 2 blocks to free one
  - (only one block modified under each indirect)
- RANDOM READS!
  - 200 IOPS, 8K block size -> free 0.8 MB/s
  - Can write at ~200MB/s
Snapshot Deletion

- Keep track of no-longer-referenced ("dead") blocks
- Each dataset (snapshot & filesystem) has "dead list"
  - On-disk array of block pointers (BP’s)
  - Blocks ref’ed by prev snap, not ref’ed by me

Snap 1 -> Snap 2 -> Snap 3 -> Filesystem

-> Snapshot Timeline ->
Snapshot Deletion

- Traverse next snap’s deadlist
- Free blocks with birth > prev snap
Snapshot Deletion (rabbit)

- O(size of next’s deadlist)
  - = O(# blocks deleted before next snap)
  - Similar to 🐢 (# deleted ~=# created)
- Deadlist is compact!
  - 1 read = process 1024 BP’s
  - Up to 2048x faster than Algo 1!
- Could still take a long time to free nothing
Snapshot Deletion

- Divide deadlist into sub-lists based on birth time
- One sub-list per earlier snapshot
  - Delete snapshot: merge FS’s sublists
**Snapshot Deletion**

- Iterate over sublists
- If mintxg > prev, free all BP’s in sublist
- Merge target’s deadlist into next’s
  - Append sublist by reference -> $O(1)$
Snapshot Deletion ( )

- Deletion: $O(\# \text{ sublists} + \# \text{ blocks to free})$
  - 200 IOPS, 8K block size -> free 1500MB/sec
- Optimal: $O(\# \text{ blocks to free})$
- $\# \text{ sublists} = \# \text{ snapshots present when snap created}$
- $\# \text{ sublists} \ll \# \text{ blocks to free}$
Talk overview

- History
- Overview of the ZFS storage system
- How ZFS snapshots work
- **ZFS on-disk structures**
- How ZFS space allocation works
- How ZFS RAID-Z works
- Future work
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>vdev1</td>
<td></td>
<td>asize</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>G</td>
<td>offset1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>vdev2</td>
<td></td>
<td>asize</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>G</td>
<td>offset2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>vdev3</td>
<td></td>
<td>asize</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>G</td>
<td>offset3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>BDX lvl type cksum E comp psize lsize</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>spare</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>spare</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>physical birth time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>logical birth time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>fill count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>checksum[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>checksum[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>checksum[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>checksum[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Talk overview

- History
- Overview of the ZFS storage system
- How ZFS snapshots work
- ZFS on-disk structures
- **How ZFS space allocation works**
- How ZFS RAID-Z works
- Future work
Built-in Compression

- Block-level compression in SPA
  - Transparent to other layers
  - Each block compressed independently
  - All-zero blocks converted into file holes

Choose between LZ4, gzip, and specialty algorithms
Space Allocation

- Variable block size
  - Pro: transparent compression
  - Pro: match database block size
  - Pro: efficient metadata regardless of file size
  - Con: variable allocation size
- Can’t fit all allocation data in memory at once
  - Up to ~3GB RAM per 1TB disk
- Want to allocate as contiguously as possible
On-disk Structures

- Each disk divided into ~200 “metaslabs”
  - Each metaslab tracks free space in on-disk spacemap
- Spacemap is on-disk log of allocations & frees

<table>
<thead>
<tr>
<th>Alloc</th>
<th>Free</th>
<th>Alloc</th>
<th>Alloc</th>
<th>Free</th>
<th>Alloc</th>
<th>Alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 10</td>
<td>0 to 10</td>
<td>4 to 7</td>
<td>2 to 2</td>
<td>5 to 7</td>
<td>8 to 10</td>
<td>1 to 1</td>
</tr>
</tbody>
</table>

- Each spacemap stored in object in MOS
- Grows until rewrite (by “condensing”)
Allocation

- Load spacemap into allocatable range tree
- Range tree is in-memory structure
  - Balanced binary tree of free segments, sorted by offset
    - So we can consolidate adjacent segments
  - 2nd tree sorted by length
    - So we can allocate from largest free segment
Writing Spacemaps

● While syncing TXG, each metaslab tracks
  ○ allocations (in the *allocating* range tree)
  ○ frees (in the *freeing* range tree)
● At end of TXG
  ○ append alloc & free range trees to space_map
  ○ clear range trees
● Can free from metaslab when not loaded
● Spacemaps stored in MOS
  ○ Sync to convergence
**Condensing**

- Condense when it will halve the # entries
  - Write `allocatable` range tree to new SM

<table>
<thead>
<tr>
<th>Alloc</th>
<th>Free</th>
<th>Alloc</th>
<th>Alloc</th>
<th>Free</th>
<th>Alloc</th>
<th>Alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 10</td>
<td>0 to 10</td>
<td>4 to 7</td>
<td>2 to 2</td>
<td>5 to 7</td>
<td>8 to 10</td>
<td>1 to 1</td>
</tr>
</tbody>
</table>

Diagram:
- 3 to 3
- 0 to 0
- 5 to 7
- Allocations and frees indicated in the diagram.
Talk overview

- History
- Overview of the ZFS storage system
- How ZFS snapshots work
- ZFS on-disk structures
- How ZFS space allocation works
- How ZFS RAID-Z works
- Future work
Traditional RAID (4/5/6)

- Stripe is physically defined
- Partial-stripe writes are awful
  - 1 write -> 4 i/o’s (read & write of data & parity)
  - Not crash-consistent
    - “RAID-5 write hole”
    - Entire stripe left unprotected
      - (including unmodified blocks)
    - Fix: expensive NVRAM + complicated logic
RAID-Z

- Single, double, or triple parity
- Eliminates “RAID-5 write hole”
- No special hardware required for best perf
RAID-Z: no partial-stripe writes

- Always consistent!
- Each block has its own parity
- Odd-size blocks use slightly more space
- Single-block reads access all disks :-(

<table>
<thead>
<tr>
<th>LBA</th>
<th>Disk</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>P₀</td>
<td>D₀</td>
<td>D₂</td>
<td>D₄</td>
<td>D₆</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>P₁</td>
<td>D₁</td>
<td>D₃</td>
<td>D₅</td>
<td>D₇</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>P₀</td>
<td>D₀</td>
<td>D₁</td>
<td>D₂</td>
<td>P₀</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>D₀</td>
<td>D₁</td>
<td>D₂</td>
<td>P₀</td>
<td>D₀</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>P₀</td>
<td>D₀</td>
<td>D₄</td>
<td>D₈</td>
<td>D₁₁</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>P₁</td>
<td>D₁</td>
<td>D₅</td>
<td>D₉</td>
<td>D₁₂</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>P₂</td>
<td>D₂</td>
<td>D₆</td>
<td>D₁₀</td>
<td>D₁₃</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>P₃</td>
<td>D₃</td>
<td>D₇</td>
<td>P₀</td>
<td>D₀</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>D₁</td>
<td>D₂</td>
<td>D₃</td>
<td>X</td>
<td>P₀</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>D₀</td>
<td>D₁</td>
<td>X</td>
<td>P₀</td>
<td>D₀</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>D₃</td>
<td>D₆</td>
<td>D₉</td>
<td>P₁</td>
<td>D₁</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>D₄</td>
<td>D₇</td>
<td>D₁₀</td>
<td>P₂</td>
<td>D₂</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>D₅</td>
<td>D₈</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Talk overview

- History
- Overview of the ZFS storage system
- How ZFS snapshots work
- ZFS on-disk structures
- How ZFS space allocation works
- How ZFS RAID-Z works
- Future work
Future work

- Easy to manage on-disk encryption
- **Channel programs**
  - Compound administrative operations
- **Vdev spacemap log**
  - Performance of large/fragmented pools
- **Device removal**
  - Move allocated space to other disks
Further reading

http://www.open-zfs.org/wiki/Developer_resources
Specific Features

- Space allocation [video (slides)] - Matt Ahrens ‘01
- Replication w/ send/receive [video (slides)]
  - Dan Kimmel ‘12 & Paul Dagnelie
- Caching with compressed ARC [video (slides)] - George Wilson
- Write throttle blog [1 2 3] - Adam Leventhal ‘01
- Channel programs [video (slides)]
  - Sara Hartse ‘17 & Chris Williamson
- Encryption [video (slides)] - Tom Caputi
- Device initialization [video (slides)] - Joe Stein ‘17
- Device removal [video (slides)] - Alex Reece & Matt Ahrens
Further reading: overview

- Design of FreeBSD book - Kirk McKusick
- Read/Write code tour video - Matt Ahrens
- Overview video (slides) - Kirk McKusick
- ZFS On-disk format pdf - Tabriz Leman / Sun Micro
Community / Development

- History of ZFS features [video](#) - Matt Ahrens
- Birth of ZFS [video](#) - Jeff Bonwick
- OpenZFS founding [paper](#) - Matt Ahrens
http://openzfs.org

Matt Ahrens
Principal Engineer, Delphix
ZFS co-creator
Brown CS 2001