What is performance analysis and tuning?
Going from this
To this
Analyzing and benchmarking performance

- Benchmarking a filesystem or block device usually mean running synthetic workload generators like FIO or Vdbench
  - Synthetic workload generators try to approximate a production workload
  - Usually done at the POC stage of deployment
- When analyzing performance in production a synthetic benchmark doesn’t help, we need performance monitoring tools
  - arcstat
  - zilstat (needs DTrace so not supported on Linux)
  - zpool iostat
  - dstat --zfs-zil --zfs-arc --zfs-l2arc (Linux only, not part of dstat package)
  - DTrace Tool Kit (DTT)
Quantifying Storage Performance

• What do we need to know to design a system to perform at some level?
  • Ideally we want the “workload” parameters
    • Average transfer size (aka block size or xfersize)
    • R/W mix
    • Sync or Async writes
    • Access pattern - mostly sequential or random
    • Expected utilization - thread count, queue lengths, etc (rarely known)
• These workload parameters aren’t always provided by vendors making it hard to design a system based on vendor quoted numbers alone
The need to verify vendor provided numbers

- In addition vendor quoted numbers may be
  - Simply inflated
  - Obtained with specialized “benchmarking” HW
  - Obtained with a synthetic benchmark that is unrealistically tuned for performance (working set always fits into cache for example)
  - Impossible to sustain over time

- Bottom line is we need to verify performance or find a resource that provides performance numbers we can trust
Common performance drainers

- Workload vs. HW capability mismatch
- HW going bad - disk, controller, loose SAS cable etc
- Highly uneven space utilization on vdevs or pool almost full
- Fragmentation - “zpool get fragmentation pool_name”
- Background deletion - “zpool get freeing pool_name”
- Silly pool geometry - mixing RAIDZs, mirrors and drive types
USE - Performance Analysis approach

- USE - a methodology for analyzing system performance by Brendan Gregg
  - Utilization - e.g. busy %
  - Saturation - e.g. queue length
  - Errors
Common OpenZFS tuning knobs
Recordsize / Volblocksize

- Defines the largest block that can be written to the dataset or zvol
- Is the unit that ZFS compresses and checksums
- `zfs get recordsize pool_name/fs`
  - 128k default
  - If changed will affects only new writes
    - `zfs set recordsize=32k pool_name/fs`
- `zfs get volblocksize pool_name/zvol`
  - Is a block device that is commonly shared through iSCSI or FC
  - 8k default and is set at creation time
  - Cannot change after creation
Recordsize / Volblocksize tuning

- Create a dataset / zvol for each application or for applications working with the same data
- For random workloads try to have recordsize/volblocksize match the average transfer size of your application(s)
  - Reduces wasting of bandwidth when recordsize >> xfersize
  - Reduces metadata overhead when recordsize << xfersize
ARC and L2ARC

- Adaptive Replacement Cache is ZFS’s much celebrated in-RAM read cache
  - Caches both MRU and FRU blocks
  - Dynamically balances between caching more FRU or more MRU blocks
  - Sometimes it’s useful to put a cap on the ARC’s maximum size with zfs_arc_max
  - In general the more RAM you have the better the read performance will be
  - Observe utilization with ”arcstat.pl/.py” or “dstat --zfs-arc”
- L2ARC is the Level 2 ARC that can be added per pool
  - It’s usually a single SSD device
  - Currently doesn’t save state on reboot
  - L2ARC isn’t free and isn’t always needed - more ARC (RAM) may be better
Why adding L2ARC device isn’t “free”

- Each block cache in the L2ARC requires a header to be added to the ARC
  - This means you’re using a bit of space in the ARC to be able to cache in L2ARC
- If you're working set size is around the size of the ARC adding an L2ARC device may hurt read performance
- OpenZFS header sizes per cached block
  - ARC only header - 176 bytes
  - ARC + L2ARC header - 208 bytes
  - L2ARC only header - 128 bytes
- Observe L2ARC utilization with "arcstat.pl/.py" or "dstat --zfs-l2arc"
- `zfs set secondarycache=<all | metadata | none> pool/dataset`
The ZFS Intent Log ensures filesystem consistency.

ZIL will satisfy the POSIX synchronous write requirement by storing write records to an on-disk log before completing the write.

ZIL is only read in an event of a crash.

Two modes for ZIL commits:

- Immediate - write user’s data into ZIL, later write into final resting place.
- Indirect - write user’s data into final resting place, ZIL gets a pointer to the data.

Don’t turn off ZIL if you are not ok with losing a TXG worth of data in the event of a power failure or crash.
• Unless a Separate Log device is added, the ZIL will be stored on the pool’s vdevs
• If SLOG is added to a pool the ZIL will be stored on these added devices
• To speed up sync writes we can choose to put the ZIL on a Separate LOG device
• Typically a SLOG will be two small mirrored latency optimized SSD devices
  • Small because it only needs to hold < 2 TXGs (zfs_dirty_data_max) worth of data
  • Mirrored since we need redundancy to protect from data loss
  • Latency optimized because the faster we can write to it the faster we can tell the application that we have it’s data on stable storage
ZIL and SLOG tuning

- Use "zilstat.ksh" (DTrace based) or "dstat --zfs-zil" (Linux kstat based) to see if a workload is sync write heavy
  - Helps determine if SLOG is needed
- Indirect vs Immediate affected by
  - logbias property
  - zfs_immediate_write_sz and zvol_immediate_write_sz (both 32k by default)
  - zil_slog_limit (1mb by default)
- SLOG can be tuned with logbias=latency (default) vs logbias=throughput
  - “throughput” setting will bypass SLOG and is useful for large sync streaming workloads
Alignment shift defines the size of the smallest block that we will send to disk
- ashift of 9 means $2^9 = 512$ bytes is the smallest block
- Currently once it’s set it can not change
- ashift should match the physical block size (PBS aka sector size) reported by the drive
- Be careful, some “Advanced Format” drives lie about their PBS
  - This means that when we send a 512 byte I/O to one of these drives, it is wasting bandwidth since it’s is working with 4k PBS internally
- Query ashift with zdb: `zdb | egrep 'ashift| name'`
RAIDZ

• RAIDZ is ZFS’s software RAID technology
  • Data + parity like standard RAID
  • Dynamic stripe width eliminating the write hole
• RAIDZ1 - single parity drive
  • Is able to deal with a single drive failure
• RAIDZ2 - double parity
  • Is able to deal with any two drives failing
• RAIDZ3 - triple parity
  • Is able to deal with 3 drives failing from a single stripe
Each (post compression) block write sent to a RAIDZ group will be spread across the drives

- Assuming a PBS of 4k and a 16k write onto a 6-wide RAIDZ2 we will write 4k of data to 4 drives and the other 2 drives will get 4k parity each
- To prevent holes, each allocation is a multiple of # of parity + 1
  - For RAIDZ1 each allocation will be a multiple of 2
  - For RAIDZ2 each allocation will be a multiple of 3
  - For RAIDZ3 each allocation will be a multiple of 4
- Consider 5-wide RAIDZ1 in the following example
  - A single square is one sector
<table>
<thead>
<tr>
<th>Color</th>
<th>Data</th>
<th>Padding</th>
<th>Parity</th>
<th>Total Written</th>
<th>% used for parity + pad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>20%</td>
</tr>
<tr>
<td>Yellow</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>Green</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>Red</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>Lavender</td>
<td>14</td>
<td>0</td>
<td>4</td>
<td>18</td>
<td>~22%</td>
</tr>
<tr>
<td>Purple</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>~33%</td>
</tr>
<tr>
<td>Cyan</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>Blue</td>
<td>11</td>
<td>0</td>
<td>3</td>
<td>14</td>
<td>~21%</td>
</tr>
</tbody>
</table>
RAIDZ performance considerations

• In general for a random access pattern workload
  • Mirror beats RAIDZ in performance
  • RAIDZ group will perform at the speed of the slowest drive in the group
  • For more IOPS - use fewer disks per group (and more groups)
  • For more usable space - use more disk per group
• Don’t use small recordsizes with devices that aren’t 512 PBS
• ashift, recordsize and RAIDZ width will define usable space availability
• Don’t worry about exact alignment since you’ll probably have compression enabled so the post compression block sizes will vary
I/O scheduler

- There are 5 I/O classes that control the number of I/Os issued to each drive
  - 1 - sync (normal demand) reads
  - 2 - async (prefetch) reads
  - 3 - sync (ZIL) writes
  - 4 - async (dirty data) writes
  - 5 - scrub / resilver
- Each class has a tunable min and max for outstanding I/Os per leaf vdev
- The scheduler will issue the min number of I/O from each class
- Then the scheduler will in the order above will try issue up to max from each class
- There is also a per vdev cap for all outstanding I/Os - zfs_vdev_max_active
I/O scheduler tuning

- Using the I/O scheduler tunables we can prioritize certain classes of I/O
- Tuning an I/O class’s max higher should lead to higher throughput and maybe higher latency for that class
- Tuning an I/O class’s min higher will make the scheduler issue more I/O before considering the class’s priority
  - Used to prioritize faster scrubs and resilvers at the expense of other I/O
Write throttle (WT)

- WT introduces artificial delays to the time it takes a TX to get assigned to a TXG
- Needed when the client application is giving us more data than we can write
  - We’ve been issuing the max from the async write I/O class but can’t keep up
- The goal is to provide consistent latency
- WT starts adding small delays when the amount of dirty data in the pool reaches 60% (by default) of the zfs_dirty_data_max
- After the 60% threshold WT becomes more and more aggressive as more dirty data is added and we stop accepting new writes when we exceed zfs_dirty_data_max
Write throttle (WT) tuning

- Observe by DTracing dmu_tx_delay or watching it in /proc/spl/kstat/zfs/dmu_tx
- Tuning zfs_dirty_data_max to be larger will allow the system to absorb bigger write spikes at the expense of having a smaller ARC
  - Will also lead to higher TXG sync times which adds latency to sync context operations like snapshotting
- WT has other knobs and tuning it is usually done per specific set of HW
Turn dedup on? Probably not

- Sounds good on paper since it decreases space utilization
- Rarely used in practice in production since it has some sharp corners as it relates to performance
- Deduped blocks are tracked in a DeDuplication Table (DDT) that essentially map a block’s checksum to a refcount and location on disk
- DDT will grow every time we write a unique (never before seen) block
- If DDT is small and cached we will be ok
- Once dedup is on to reverse it you have to copy data to a dedup=off dataset
- Only turn on dedup if you are sure the dataset will be very highly dedupable
  - Even VMs spawned from the same image will diverge over time
  - “zdb -S pool_name” will simulate dedup on an existing pool
Common dataset settings

- ARC works well, most of the time we tune for better write performance
- `zfs set atime=off pool/ds`
  - don’t modify access time to reduce write inflation
- `zfs set redundant_metadata=most pool/ds`
  - Reduces write inflation by storing less metadata. Metadata is still stored redundantly due to pool layout (e.g. RAIDZ) and the copies property
- `zfs set xattr=sa pool/ds (Linux only)`
  - Needed when using POSIX ACLs
  - Decreases the amount of disk IO required to access extended attributes
- `zfs set primarycache=metadata pool/ds`
  - May be useful when an application does it’s own read caching
## Where to tune global settings

<table>
<thead>
<tr>
<th>ZFS on Linux</th>
<th>illumos</th>
<th>FreeBSD</th>
<th>OpenZFS on OS X</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc/modprobe.d/zfs.conf</td>
<td>/etc/system</td>
<td>/etc/sysctl.conf</td>
<td>/etc/zfs/zsysctl.conf</td>
</tr>
</tbody>
</table>
# References to resources

<table>
<thead>
<tr>
<th>What</th>
<th>Where</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAIDZ width - usable space vs IOPS</td>
<td><a href="https://www.delphix.com/blog/delphix-engineering/zfs-raidz-stripe-width-or-how-i-learned-stop-worrying-and-love-raidz">https://www.delphix.com/blog/delphix-engineering/zfs-raidz-stripe-width-or-how-i-learned-stop-worrying-and-love-raidz</a></td>
</tr>
<tr>
<td></td>
<td><a href="https://blogs.oracle.com/roch/entry/when_to_and_not_to">https://blogs.oracle.com/roch/entry/when_to_and_not_to</a></td>
</tr>
<tr>
<td>I/O scheduler and write throttle</td>
<td><a href="https://www.delphix.com/blog/delphix-engineering/tuning-openzfs-write-throttle">https://www.delphix.com/blog/delphix-engineering/tuning-openzfs-write-throttle</a></td>
</tr>
<tr>
<td>On recordsize</td>
<td><a href="https://www.joyent.com/blog/bruning-questions-zfs-record-size">https://www.joyent.com/blog/bruning-questions-zfs-record-size</a></td>
</tr>
<tr>
<td>ARC stat</td>
<td><a href="http://www.c0t0d0s0.org/archives/5329-Some-insight-into-the-read-cache-of-ZFS-or-The-ARC.html">http://www.c0t0d0s0.org/archives/5329-Some-insight-into-the-read-cache-of-ZFS-or-The-ARC.html</a></td>
</tr>
</tbody>
</table>
Thank you

- Questions?!