Modern File Systems

Parallel Storage Systems

2023-05-15

Jun.-Prof. Dr. Michael Kuhn
michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

Modern File Systems

- Review
- Introduction
- Example: ZFS
- Data Reduction
- Summary

- What is the purpose of direct I/O?
  1. Disable the use of stackable file systems
  2. Circumvent the operating system's page cache
  3. Send requests directly to the storage device

- Why are file and directory names stored in the directory entry?
    1. Improve performance
    2. Allow multiple names for a file or directory
    3. No space left in the inode

- What is the benefit of using extents vs. block pointers?
  1. Less management overhead
  2. Improved performance
  3. Allow addressing larger files

- What is the purpose of a file system journal?
    1. Improve performance
    2. Guarantee consistency in case of a crash
    3. Provide redundancy in case of errors

Outline

Modern File Systems

- Review
- Introduction
- Example: ZFS
- Data Reduction
- Summary

Review

- Reminder: File systems take care of structuring storage
  - They manage data and metadata (permissions, timestamps etc.)
  - One of the most important aspects is block allocation and management
- File systems use underlying storage devices and arrays
  - Examples: Logical Volume Manager (LVM), mdadm
- File systems typically only offer rudimentary functions
  - Creating, deleting, reading and writing files and directories
  - Storage devices and arrays have to be managed separately

- Requirements for file systems keep growing
    - Data integrity to be able to access data in the future
    - Storage management for large storage systems
    - Convenience functions to simplify workflows
- Error rates for SATA HDDs are around 1 in 10^{14} to 10^{15} bits [Seagate, 2020]
    - That is, one bit error happens every 12.5–125 TB
    - Additional errors may occur in RAM, the controller, cables, the driver etc.
- Error rate can be problematic especially with today's HDD capacities
    - These data volumes are reachable even in daily use
    - Bit errors can also happen in important data structures such as the superblock

Overview…

- File system usually does not have knowledge about the storage array
    - Storage array also does not know about file system contents
    - Even block allocation status is unknown without TRIM/DISCARD
- Mutual knowledge can be important to achieve optimal performance
    - For instance, ext4 offers special options: `-E stride=n,stripe_width=m`
        - `stride` denotes the number of file system blocks per storage device
        - `stripe_width` denotes the number of file system blocks per stripe
- Reconstruction times are high due to missing knowledge about contents
    - Reconstruction can take $\geq$ 20 h with today's HDD capacities

Modern Features

- Snapshots to easily go back to a previous state
  - Snapshots can also be used to handle checkpoints efficiently

Modern Features

- Snapshots to easily go back to a previous state
    - Snapshots can also be used to handle checkpoints efficiently
- Sub file systems to partition the namespace
    - Allows separating different types of data or using different configurations

Modern Features

- Snapshots to easily go back to a previous state
  - Snapshots can also be used to handle checkpoints efficiently
- Sub file systems to partition the namespace
  - Allows separating different types of data or using different configurations
- Compression and deduplication to reduce the amount data
  - Both techniques can improve throughput and capacity

Modern Features

- Snapshots to easily go back to a previous state
  - Snapshots can also be used to handle checkpoints efficiently
- Sub file systems to partition the namespace
  - Allows separating different types of data or using different configurations
- Compression and deduplication to reduce the amount data
  - Both techniques can improve throughput and capacity
- Encryption to keep data safe
  - Especially important in industry but also for important research

Modern Features

- Snapshots to easily go back to a previous state
  - Snapshots can also be used to handle checkpoints efficiently
- Sub file systems to partition the namespace
  - Allows separating different types of data or using different configurations
- Compression and deduplication to reduce the amount data
  - Both techniques can improve throughput and capacity
- Encryption to keep data safe
  - Especially important in industry but also for important research
- Efficient backups that do not require scanning the whole namespace
  - Storage systems can easily reach petabytes of data

Outline

Modern File Systems

- Review
- Introduction
- Example: ZFS
- Data Reduction
- Summary

Overview

- ZFS is used as an example here
  - The basic principles can be found in other modern file systems as well
  - Further examples: btrfs, bcachefs, Stratis etc.
- ZFS is a local meta file system
  - It was previously called the Zettabyte File System, today just ZFS
  - ZFS contains an integrated volume manager and much more
- It has been initially developed by Sun Microsystems
  - 2001: Start of development
  - 2005: First publication in OpenSolaris
  - 2006: First publication in Solaris 10
  - 2008: ZFS-based appliances
  - 2010: Oracle ends development as open source

Overview...

- Recent developments
  - 2010: Separation of illumos
  - 2013: Start of the OpenZFS initiative
  - 2013: Supported as a backend file system in Lustre
  - 2019: Reintegration of ZFS on Linux into OpenZFS
- Many operating systems are supported
  - Solaris: Closed source, incompatible with OpenZFS
  - OS X: OpenZFS on OS X (O3X)
  - FreeBSD: Full support
  - Linux: ZFS on Linux (many distributions)
- CDDL and GPL are incompatible
  - Full integration into Linux is therefore complicated

Compatibility

- Both the file system's functionality an its on disk format are versioned
  - Sun and Oracle used an incrementing version number
  - Compatibility is limited due to Oracle continuing development as closed source
- OpenZFS uses a development model with feature flags
  - The version was pinned to 1000 or 5000
  - Examples: `async_destroy`, `lz4_compress`, `embedded_data` and `large_blocks`
    - `async_destroy`: File systems are destroyed in the background
    - `lz4_compress`: Makes available lz4 as a compression algorithm
    - `embedded_data`: Files that are smaller than 112 bytes can be stored in the block pointer
    - `large_blocks`: Blocks can get larger than 128 KiB

Features

- ZFS is the first 128 bit file system
    - 64 bits are sufficient for adressing 16 EiB
    - 128 bits cannot be utilized fully
    - *“Populating 128-bit file systems would exceed the quantum limits of earth-based storage. You couldn’t fill a 128-bit storage pool without boiling the oceans.”*
        - Jeff Bonwick, former ZFS head of development
- Data integrity is an essential feature
    - Errors are detected and repaired automatically
- Easy administration paired with high performance
    - Administration only requires two tools

Regression Tests

- Reading, writing, creating and deleting files and directories
- Creating and destroying file systems and pools
- Enabling and disabling compression
- Changing the checksum algorithm
- Adding and removing devices
- Changing the caching and scheduling strategies
- Writing random data to one half of a mirror
- Simulating crashes

"Probably more abuse in 20 seconds than you'd see in a lifetime."

– Jeff Bonwick, former ZFS head of development

Motivation

- Traditional file systems often use outdated concepts
- No protection against data corruption
  - ext4 can only save checksums for metadata
- High administration overhead
  - Storage devices have to be grouped into storage arrays
  - Devices and arrays have to be partitioned
  - Partitions have to be formatted
- Traditional concepts are often inflexible
  - Static block and file system sizes
  - Often static file and directory counts

Limits

- Maximum number of objects per directory: 2^{48}
  - 2^{32} for ext4 (per file system)
- Maximum size of a file: 16 EiB (2^{64} bytes)
  - 16 TiB for ext4
- Maximum size of a pool: 256 ZiB (2^{78} bytes)
  - 64 ZiB for ext4
- Maximum number of devices per pool: 2^{64}
- Maximum number of pools: 2^{64}
- Maximum number of file systems per pool: 2^{64}

Design Principles

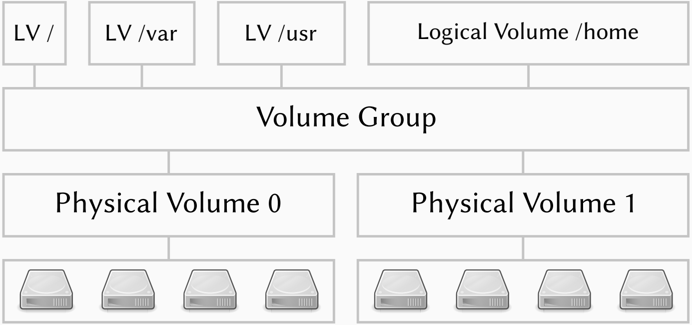
- Pools
  - No manual management of HDDs, partitions etc. anymore
  - Pool provides storage space for all file systems
- Data integrity
  - Has been deemed too expensive in the past
  - CPUs have enough computation power reserves nowadays
- Transactions
  - Data is always consistent (like in a database)
  - Time-consuming file system checks can be skipped

Pools

- Traditionally, there is one file system per partition
    - Volume managers allow spanning a file system across multiple devices
        - It is also possible to use parts of a device (that is, partitions)
- Current file systems are very static
    - Changing their size or the size of their data structures can be problematic
- ZFS introduces a new pool concept
    - Goal: Using the total capacity and throughput of the available hardware
    - Keeps file systems dynamic by outsourcing storage allocation

Pools…

- Traditional architecture
  - RAID, LVM, file system
  - Worst case: Three technologies
- File systems are created on top of logical volumes



Pools…

```
$ mdadm --create /dev/md0 --level=5 --raid-devices=4 /dev/sd[abcd]
$ mdadm --create /dev/md1 --level=5 --raid-devices=4 /dev/sd[efgh]
```

Pools...

```
$ mdadm --create /dev/md0 --level=5 --raid-devices=4 /dev/sd[abcd]
$ mdadm --create /dev/md1 --level=5 --raid-devices=4 /dev/sd[efgh]
```

```
$ pvcreate /dev/md0
$ pvcreate /dev/md1
$ vgcreate tank /dev/md0 /dev/md1
$ lvcreate --size 15G --name root tank
$ lvcreate --size 25G --name var tank
$ lvcreate --size 30G --name usr tank
$ lvcreate --size 75G --name home tank
```

Pools...

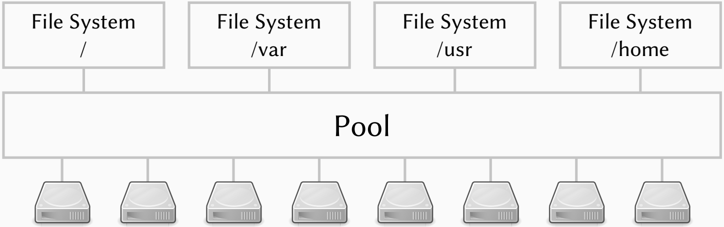
```
$ mdadm --create /dev/md0 --level=5 --raid-devices=4 /dev/sd[abcd]
$ mdadm --create /dev/md1 --level=5 --raid-devices=4 /dev/sd[efgh]
```

```
$ pvcreate /dev/md0
$ pvcreate /dev/md1
$ vgcreate tank /dev/md0 /dev/md1
$ lvcreate --size 15G --name root tank
$ lvcreate --size 25G --name var tank
$ lvcreate --size 30G --name usr tank
$ lvcreate --size 75G --name home tank
```

```
$ mkfs.ext4 /dev/mapper/tank-root
$ mkfs.ext4 /dev/mapper/tank-var
$ mkfs.ext4 /dev/mapper/tank-usr
$ mkfs.ext4 /dev/mapper/tank-home
```

Pools...

- ZFS pool architecture
  - ZFS takes care of all three layers
- File systems allocate space in the pool



Pools…

```
$ zpool create tank raidz /dev/sd[abcd] raidz /dev/sd[efgh]
$ zfs create tank/root
$ zfs create tank/var
$ zfs create tank/usr
$ zfs create tank/home
```

Pools…

- Pools consist of virtual devices (vdevs)
  - Data is distributed across all vdevs dynamically
- Virtual devices can be real devices or arrays of those
  - Mirror (RAID 1), RAID-Z (RAID 5), RAID-Z2 (RAID 6), RAID-Z3
  - Recently added: Distributed RAID (dRAID)
- It is not possible to reproduce all RAID levels
  - For example, it is not possible to create a RAID 51 array
  - RAID 10, RAID 50 and RAID 60 are possible
- ZFS's RAIDs do not suffer from the write hole problem
  - Reminder: The write hole can occur between writing the data and the parity

Pools…

- ZFS also supports so-called volumes
  - Volumes are exported as block devices
  - Can be used to use traditional file systems etc. within a pool
- All pool functionality can be used for volumes
  - Snapshots, compression etc. are supported transparently

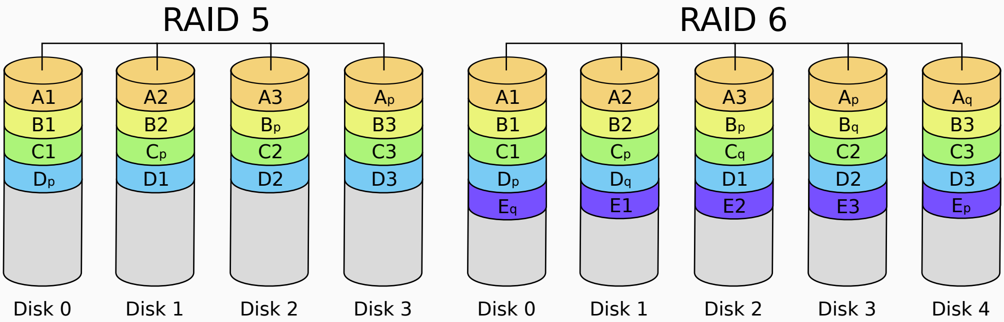
```
$ zfs create -V 4G tank/swap
$ zfs create -V 75G tank/home
$ mkswap /dev/zvol/tank/swap
$ mkfs.ext4 /dev/zvol/tank/home
```

Dynamic Striping

- ZFS intelligently distributed data across all virtual devices
- Multiple selection criteria are used
    - Capacity
    - Performance (latency, throughput, utilization)
    - Status (for instance, mirror with failed device)
- New virtual devices are used automatically
    - Existing data is not rebalanced
    - New device is preferred to match usage

Dynamic Striping...

1. Virtual device selection
    - Prefer new or empty virtual devices
    - Avoid degraded virtual devices
    - Finally, use a round-robin approach
        - More striping methods can be added in the future
2. Metaslab selection
    - Prefer the outer regions of HDDs because they are faster
    - Prefer metaslabs that have already been used
3. Block selection
    - Choose the first block with enough free space
        - More algorithms can be added in the future



[Wikipedia, 2021]

- Data and parity have to be updated
  - Failures in between can cause the write hole
  - Operations across multiple HDDs have to be performed atomically
- Writing partial stripes is inefficient
  - Read-modify-write: Two reads and two writes
- Solution: Hardware controllers with large caches and uninterruptible power supply
  - The original idea was a Redundant Array of *Inexpensive* Disks

- The write hole can be eliminated using copy on write and transactions
  - This combination allows atomic updates of data and parity
  - Normal HDDs are enough for this to work
- ZFS does not write partial stripes
  - Each block is in its own stripe
  - This improves performance but makes reconstruction more complicated
    - Requires the RAID layer to know about the file system structure

- Error scenario on a mirror with two HDDs
    1. Application reads data
    2. ZFS reads data from the first HDD
    3. ZFS detects that the data is incorrect (due to its checksum)
    4. ZFS reads the data copy from the second HDD
    5. ZFS detects that the data is correct
    6. The incorrect data is overwritten with the correct one
    7. Data is forwarded to the application
- In traditional file systems, steps 3–6 do not exist
    - ZFS can also detect when both copies of the data are incorrect

- Traditional RAID systems can only detect errors but not correct them
  - This would require reading and comparing the parity for each access
  - It remains unclear whether the data or the parity is incorrect
- Normally, incorrect data is forwarded to the application
  - This can cause crashes but also silent data corruption later on
- Detecting and correcting incorrect data is very important
  - Data can be very expensive to compute and store
  - Depending on its size, backups might be infeasible

- Reconstruction can be handled intelligently in ZFS
- Traditionally, whole devices have to be reconstructed
    - This is due to the strict separation of storage array and file system
    - Reconstruction is done by performing a block-wise XOR over the devices
    - It is not possible to check for correctness using only parity
- ZFS only needs to consider blocks actually used by the file system
    - For temporary failures, only changes have to be considered
    - Reconstruction is safer due to top-down reconstruction of the tree
        - Losing inner nodes is fatal, while data blocks can be handled more easily
        - A missing inner node makes the whole sub-tree inaccessible

- Scrubbing allows finding and correcting errors
- During a scrub, the following operations are performed for each block
    1. Block and its checksum are read
    2. Block is checksummed and result is compared to stored checksum
    3. If the checksum is incorrect, try recovering the block
- Scrubbing is typically not performed automatically
    - Developers recommend weekly or monthly scrubs
    - Errors can only be detected when data is accessed

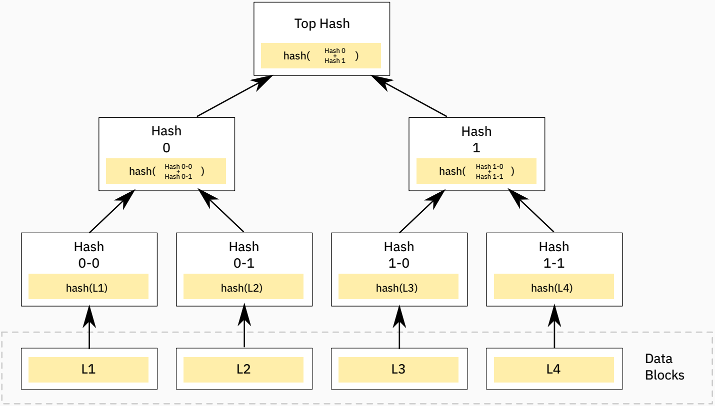
Transactions

- All operations are performed within transactions
    - File system level: All modifications done to files and directories
    - Storage system level: Transactions are combined into transaction groups
- Transactions allow ZFS to be always consistent
    - No journaling is required, reducing overhead
    - No file system checks are necessary after a crash

File System Structure

- ZFS is realized as a hash tree of blocks
    - This is also called a Merkle tree
- Each block contains a checksum
    - There is a range of different algorithms available
- Data integrity is checked with each read operation
- Multiple copies of metadata are stored at all times
    - Allows reconstructing metadata even without a RAID
    - Metadata is small but important

- Leaf nodes contain checksums of data blocks
- Inner nodes contain checksums of their children

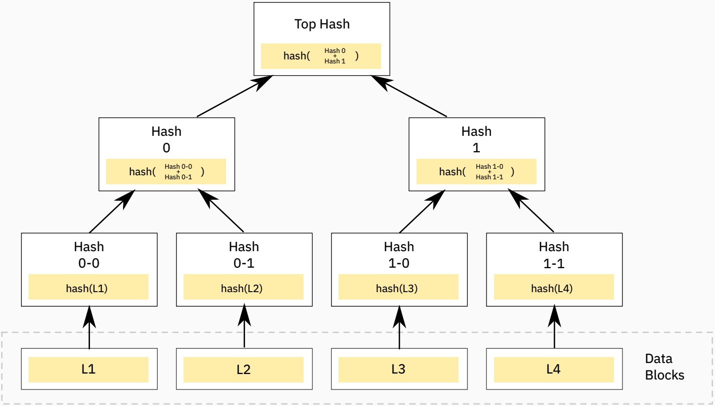


[Azaghal, 2012]

Copy on Write

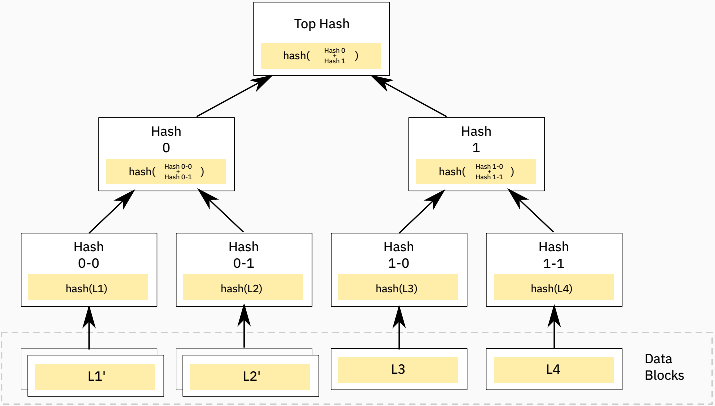
- Traditionally, blocks are modified in-place
  - This can lead to inconsistencies if these updates are interrupted
- Using copy on write, blocks are never overwritten but instead copied
  - The original is read, a copy is modified and written at another location
  - To be precise, this is redirect on write but typically called copy on write
- All changes are done outside the live file system structure
  - If the system crashes, modifications are simply not visible and can be discarded
- In a final step, new blocks are integrated atomically

1. Initial state
   - Each node contains checksum of children
   - Data blocks L1 and L2 should be updated



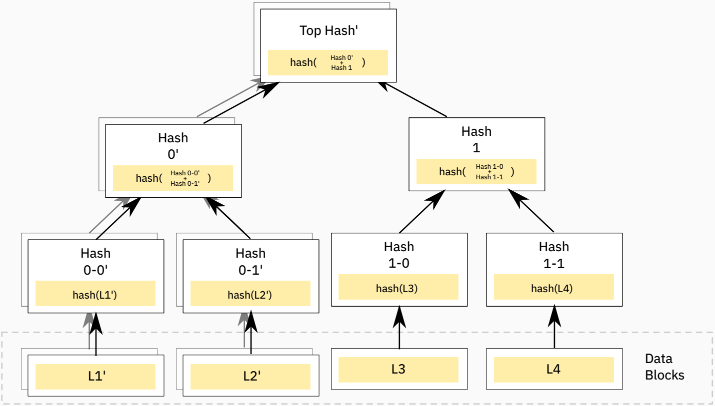
[Azaghal, 2012]

2. New blocks are allocated
    - Original data is read, modified and written as L1’ and L2’



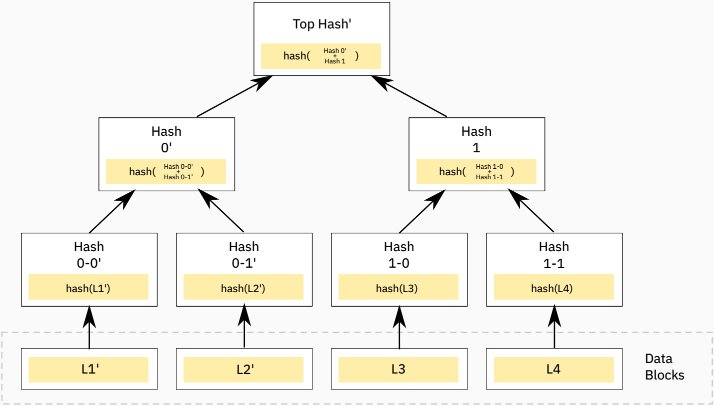
[Azaghal, 2012]

3. New blocks are allocated
    - Inner nodes also use copy on write
    - Pointers are set to new blocks, up to the root



[Azaghal, 2012]

4. Root node is updated
   - Root update has to happen atomically



[Azaghal, 2012]

Quiz

- How would you update the root node atomically?
    1. Storage devices guarantee atomic updates
    2. Use direct I/O
    3. Make sure to write 512 bytes at once
    4. Make sure to write 4,096 bytes at once

- ZFS's root node is called the uberblock
- ZFS uses an uberblock array with 128 entries
  - Replicas of the array are stored at different points of the pool
  - The array is used in a round-robin fashion
- Uberblock entries contain a transaction ID and a checksum
  - When mounting, the uberblock entry with the highest transaction ID is used
  - Uberblock integrity is ensured using checksum

Snapshots and Clones

- Snapshots simplify a wide range of use cases
  - Making older data available (for instance, using daily snapshots)
  - Multiple checkpoints within a file to avoid redundancies
    - ZFS's snapshots are too coarse-grained for this
- Snapshots are very easy due to the copy on write scheme
  - Taking a snapshot means keeping the old root node around
  - Old pointers and blocks are not deleted due to reference counting
- ZFS's snapshots can only be read
  - Snapshots can be found in the `.zfs` directory
  - This concept allows easy access to snapshots for users

Snapshots and Clones…

- The file system can also be rolled back to an earlier snapshot
    - Simply add a new uberblock entry for the old tree
    - This discards all changes since the snapshot (after a while)
- Mutable snapshots are called clones in ZFS
    - Unmodified blocks are shared (as with snapshots)
    - Changes are integrated using the regular copy on write scheme
- Clones are almost as easy to realize as snapshots
    - Only actual changes require additional storage space due to copy on write

Backups

- Backups are problematic for large storage systems
  - Traditional tools have to scan the whole namespace for changes
  - Snapshots allow more efficient approaches for backups
- Full backups
  - A snapshot can be replicated onto another system
- Incremental backups
  - Instead of scanning for changes, use difference between two snapshots
  - Overhead depends on the amount of changes between the snapshots
- This also allows easy replication schemes
  - Example: Create snapshots every minute and transfer them to another system

Outline

Modern File Systems

- Review
- Introduction
- Example: ZFS
- Data Reduction
- Summary

Compression

- Data reduction is becoming increasingly important
    - Storage throughput and capacity do not improve at the same rate as computation
- ZFS supports transparent compression
    - Can be enabled and disabled on a file system level
    - Supports multiple compression algorithms
    - Currently, zle, gzip, lzjb, lz4 and zstd are available
- Compression is currently static
    - The selected compression algorithm is used for all data
    - Research topic: Adaptive and dynamic compression

Compression...

- zle eliminates sequences of zeroes
  - zle stands for zero-length encoding
  - Typically only achieves very low compression ratios
  - ZFS always enables zle when compression is active
- gzip achieves good compression ratios but is slow
  - gzip supports several compression levels (1–9)
  - Even fast levels are relatively slow ($\approx$ 50 MB/s)
  - Decompression is much faster than compression ($\approx$ 300 MB/s)

Compression…

- lzjb has been developed specifically for ZFS
  - LZ: Lempel Ziv
  - JB: Jeff Bonwick (former ZFS head of development)
  - Promises high performance to avoid slowing down I/O
- lz4 is a standard algorithm and faster than lzjb
  - Delivers high compression throughput ($\approx$ 600 MB/s)
  - Decompression throughput is even higher ($\approx$ 3 GB/s)
- zstd is another standard algorithm, designed by the lz4 creator
  - Delivers compression ratios comparable to gzip at much higher speeds

Deduplication

- Deduplication is another data reduction technique
  - Data is split up into blocks (statically or dynamically)
  - Redundant blocks are only stored once and referenced otherwise
  - Duplicates are identified according to their checksum
- Data integrity is an important factor when using deduplication
  - Data might differ even if the checksum is the same
  - ZFS changes the checksum algorithm to SHA256 when enabling deduplication
  - Optionally, data can be compared byte-for-byte, reducing performance

Deduplication...

- Deduplication requires additional storage space
  - Blocks and their checksums have to be kept in a separate table
  - For each write operation, this table has to be checked
  - Table should be kept in main memory for fast access
- Deduplication ratio is very dependent on the chosen block size
  - Large blocks reduce the deduplication ratio significantly
  - Smaller blocks increase storage requirements of the table
- Rule of thumb: 10+ GB per TB of data with 8 KiB blocks

Quiz

- Which data reduction technique would you use for a 1 PB file system?
    1. Deduplication
    2. Zero-length encoding
    3. Compression
    4. None

Performance Overhead

- Data integrity and convenience features introduce some overhead
  - Checksums have to be computed, copy on write requires read-modify-write, compression/deduplication/encryption require additional computation
- ZFS uses a pipeline scheduler
  - Each operation has a priority and a deadline
  - Read operations have higher priorities than write operations
- Operations can be merged and sorted for performance
  - Enables efficient copy on write, otherwise sub-tree copies for each operation

Performance Overhead…

```
$ openssl speed sha256
type    16 bytes    64 bytes    256 bytes    1024 bytes   8192 bytes   16384 bytes
sha256 271653.03k 674761.22k 1381621.50k 1877547.15k 2087605.59k 2111020.86k
```

Performance Overhead...

```
$ openssl speed sha256
type    16 bytes   64 bytes   256 bytes   1024 bytes  8192 bytes  16384 bytes
sha256 271653.03k 674761.22k 1381621.50k 1877547.15k 2087605.59k 2111020.86k
```

```
$ openssl speed aes-256-cbc
type         16 bytes    64 bytes    256 bytes   1024 bytes 8192 bytes 16384 bytes
aes-256-cbc 218166.47k 228868.11k 228862.38k 231104.47k 230176.09k 231321.26k
```

Performance Overhead…

```
$ openssl speed sha256
type    16 bytes    64 bytes    256 bytes    1024 bytes   8192 bytes   16384 bytes
sha256 271653.03k 674761.22k 1381621.50k 1877547.15k 2087605.59k 2111020.86k
```

```
$ openssl speed aes-256-cbc
type         16 bytes    64 bytes    256 bytes   1024 bytes 8192 bytes 16384 bytes
aes-256-cbc 218166.47k 228868.11k 228862.38k 231104.47k 230176.09k 231321.26k
```

```
$ openssl speed -evp aes-256-cbc
type         16 bytes     64 bytes     256 bytes    1024 bytes   8192 bytes   16384 bytes
aes-256-cbc 1015346.03k 1122795.31k 1150612.05k 1160046.45k 1160324.20k 1156639.40k
```

Performance Overhead…

```
$ time gzip -9 -v fonts.tar
fonts.tar: 43.9%
16,75s user 0,10s system 99% cpu 16,876 total (19.0 MB/s)
$ time gzip -1 -v fonts.tar
fonts.tar: 39.6%
 6,02s user 0,08s system 99% cpu  6,110 total (52.5 MB/s)
```

Performance Overhead...

```
$ time gzip -9 -v fonts.tar
fonts.tar: 43.9%
16,75s user 0,10s system 99% cpu 16,876 total (19.0 MB/s)
$ time gzip -1 -v fonts.tar
fonts.tar: 39.6%
 6,02s user 0,08s system 99% cpu  6,110 total (52.5 MB/s)
```

```
$ lz4 -b -12 fonts.tar
336517120 -> 206425036 (1.630),   22.4 MB/s, 3829.5 MB/s
$ lz4 -b -9 fonts.tar
336517120 -> 207166607 (1.624),   40.0 MB/s, 3824.9 MB/s
$ lz4 -b -1 fonts.tar
336517120 -> 244191917 (1.378),  740.3 MB/s, 4264.0 MB/s
$ lz4 -b --fast=1 fonts.tar
336517120 -> 249659973 (1.348),  867.3 MB/s, 4586.6 MB/s
$ lz4 -b --fast=12 fonts.tar
336517120 -> 275617703 (1.221), 1421.5 MB/s, 5956.4 MB/s
```

Performance Overhead...

```
$ zstd -b -19 fonts.tar
336517120 -> 154986646 (2.171),   3.13 MB/s,  632.3 MB/s
$ zstd -b -9 fonts.tar
336517120 -> 170364600 (1.975),   20.2 MB/s,  975.1 MB/s
$ zstd -b -1 fonts.tar
336517120 -> 203556107 (1.653),  395.1 MB/s, 1082.3 MB/s
$ zstd -b --fast=1 fonts.tar
336517120 -> 231137047 (1.456),  496.0 MB/s, 2335.5 MB/s
$ zstd -b --fast=9 fonts.tar
336517120 -> 253785571 (1.326),  877.3 MB/s, 3045.1 MB/s
$ zstd -b --fast=19 fonts.tar
336517120 -> 268692294 (1.252), 1122.1 MB/s, 3478.3 MB/s
```

Outline

Modern File Systems

- Review
- Introduction
- Example: ZFS
- Data Reduction
- Summary

Summary

- File systems organize data and metadata
  - Modern file systems offer data integrity, convenience, data reduction etc.
- Copy on write can help keeping the file system consistent
  - Data is never overwritten in-place, avoiding inconsistencies
- Integrated volume management has several advantages
  - Data integrity can be ensured and higher performance achieved
- Modern functionality can also be useful for parallel distributed file systems
  - Checksums are especially useful for large amounts of data
  - Transactions and snapshots allow new ways to store data

References

[Azaghal, 2012] Azaghal (2012). **Diagram of a binary hash tree.** https://en.wikipedia.org/wiki/File:Hash_Tree.svg. License: CC0 1.0.

[Seagate, 2020] Seagate (2020). **IronWolf Pro Data Sheet.** https://www.seagate.com/files/www-content/datasheets/pdfs/ironwolf-pro-18tb-DS1914-16-2011US-en_US.pdf.

[Wikipedia, 2021] Wikipedia (2021). **Standard RAID levels.** https://en.wikipedia.org/wiki/Standard_RAID_levels.