WILDFIRE STORAGE

Flash Memory Summit

Object Storage and IO Amplification

QR Code to Download this Presentation

WILDFIRE STORAGE

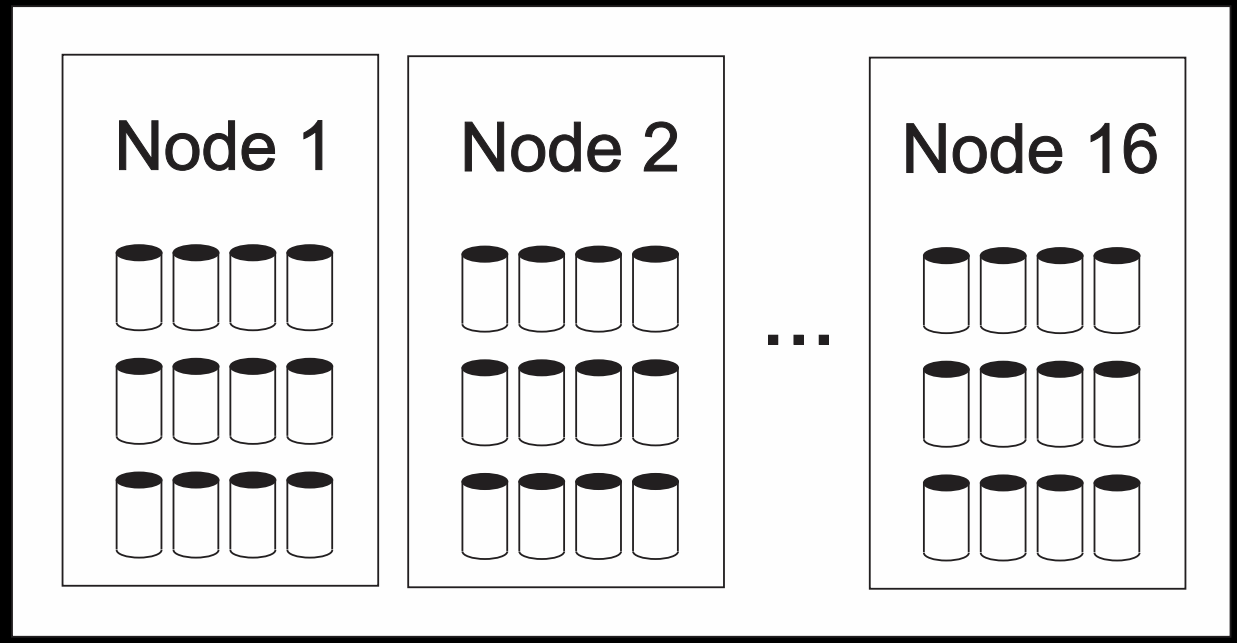
Flash Memory Summit

Object Storage and IO Amplification

Part 1 – The Anatomy of an S3 Server

An Example S3 Server – MinIO

- We are going to use a MinIO configuration as our “example” S3 server.
  - Other servers exist, but MinIO is popular, and pretty easy to analyze.
- Our MinIO Configuration
  - 16 nodes:
    - 12 20TB HDDs per node
    - 192 total disks
    - 3.8 PB raw capacity
    - 2.8 PB usable capacity
  - 12+4 erasure code config
  - XFS



MinIO IO Amplification

- Every S3 “GET” requires:
  - 12 object retrievals from 12 HDDs on 12 nodes
- Every “node get” requires:
  - One directory lookup (for the bucket)
  - One file read for the object control record
  - Additional file reads depending on the size of the object
- So we are at 12 x (1 + 1 + 1) “directory searches” for every object hit.

File System IO Amplification

- XFS is good or terrible, depending on directory size.
- Very big directories should be avoided.
  - MinIO does not have any control over bucket object count.

Directory size	IO count	KBs read	IO Amp	Read Latency (ms)
10	2	32	2.0	31
100	4	52	3.2	35
1,000	6	60	3.7	29
10,000	6	60	3.7	36
100,000	7	64	4.0	38
1,000,000	25	136	8.5	157
10,000,000	193	808	50.5	1,616
100,000,000	1,684	6,772	523.2	13,544

Total IO Amplification can get really bad.

- For buckets with large numbers of objects:
  - 100M objects – 16KB – 1.6 TB
  - x 16 x 2 files = 3.2B files
  - across 192 disks (16 nodes x 12) = 16.7M files per disk
- XFS tests at 193 IOs to read a file from a 10M file directory
  - Allowing for “some” caching, use 100 IOs for this example.

Total IO amplification can get really bad.

- @ 100 IOs to get an object from a target disk
- @ 12 disks needed to assemble an object
- 100 x 12 = 1200 “IOs” needed to get an object
- 192 disks @ 200 IOPS “available” = 38,400 IOs available
- 38,400 / 1200 = **32** Object GETs per second for the entire cluster
  - 32 x 16KB = **512** KB/sec
    - which is not even GigE
  - 512 KB/sec is 1/75,000th the bandwidth of the disks.

It is not always this bad

- RAM caches many IOs
- Most directories don’t have 1M+ files
- Even at 10 IOs per objects ...
  - 192 x 200 / 12 / 10 = 320 GETs/sec

So, this is why MinIO always benchmarks SSDs.

- Instead of 16 x 12 x 200 IOPS for HDDs, SSDs are 16 x 12 x 500,000
  - 38,400 vs 72,000,000 IOPS – 1,800X
- Even so, the IO amplification hurts.
- ... and writes are worse.

MinIO x Erasure Codes = IO Amplification

- GET – 12 x FS-Amp = 120 to 1,800 disk IOs
- PUT – 16 x FS-Amp = 160 to 2,400 disk IOs
  - Plus, overhead to track history, do billing, etc.
- We will try to mitigate some of this overhead.

WILDFIRE STORAGE

Flash Memory Summit

Object Storage and IO Amplification

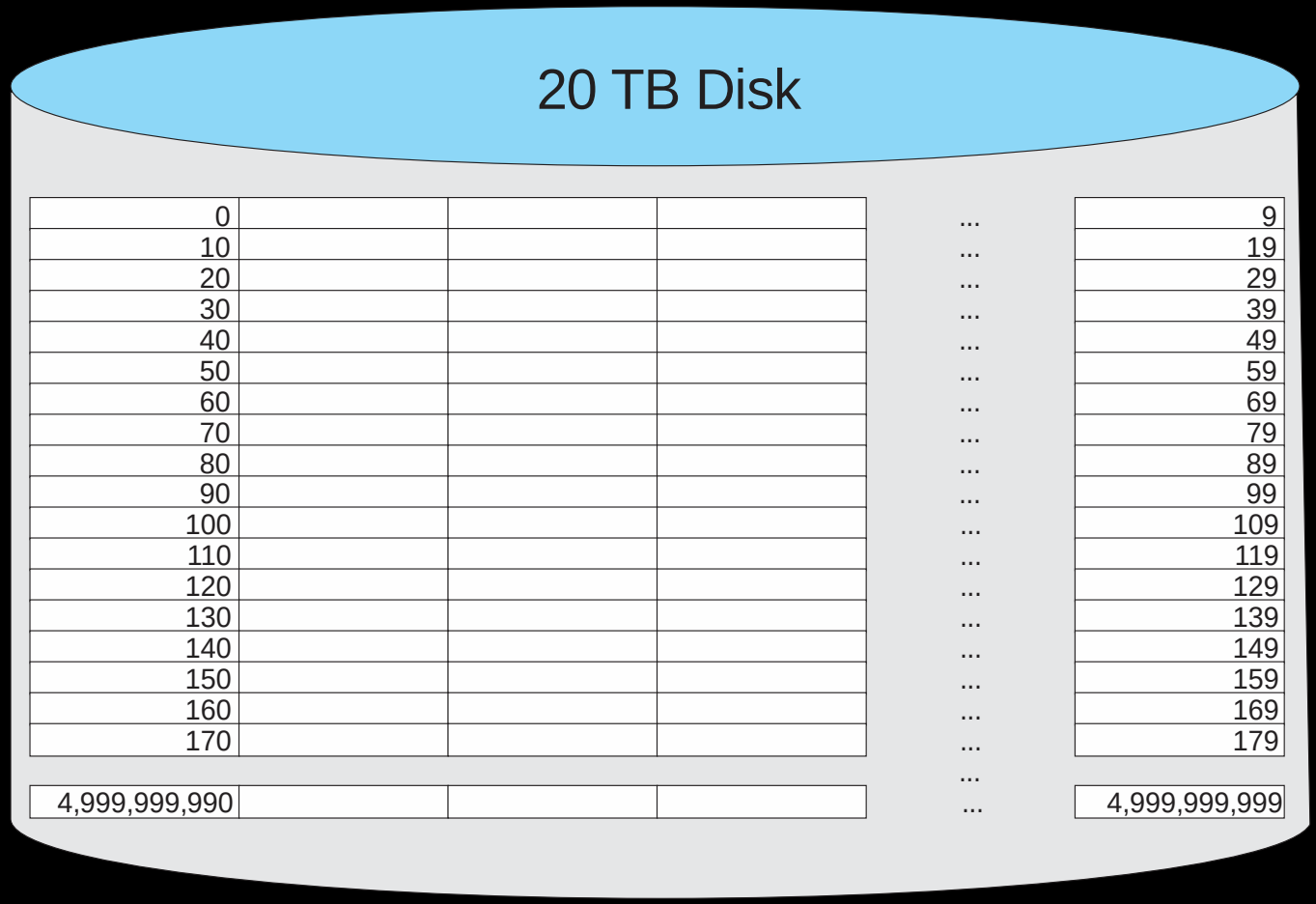
Part 2 – A File System for Objects

The Basics of File Systems

- File systems are built on top of “block devices”
  - A block device has a particular set of functionality
    - Blocks can be read
    - Blocks can be written
    - Limited control over what happens with caching after a crash
  - In addition, the block device can have a set of performance characteristics that vary from technology to technology.
    - Flash SSDs like random reads
    - Hard drives like linear IO
    - Some devices have restricted write rules
      - Zoned SSDs
      - HA-SMR / HM-SMR HDDs

All Block Devices Have ...

- A size: ... ie a 20TB hard disk.
- A fixed number of fixed sized blocks:
  - @ 4K, this is 5 billion blocks.



All Block Devices Have ...

- Rules for how writes are handled.
  - This is often quite ugly.
- Writes can be re-ordered ...
  - by the kernel
  - by the disk
- Only a single write is guaranteed intact.
- You can control basic caching with ...
  - Barriers
  - Flush

Why Crash Behavior Is Important

- The hardest part of designing a file system is handling data consistency if the server or drive crashes at an unexpected, arbitrary, and likely inconvenient time.
- The block device does not help much. Remember ...
  - Updates can get re-ordered
  - Only a single block is guaranteed to get to disk intact.
  - Barriers and flush operations are available:
    - ... but they are often very slow.

File Systems have to do a lot of heavy lifting to insure data integrity

- File systems need “atomic updates”:
  - An atomic update is where multiple blocks are updated such that either they all make it to media, or none of them do.
- Atomic updates can be implemented with:
  - Journals, update randomly, and then update again sequentially.
  - Copy on Write semantics where a single block points to a whole new set of data.
  - Generation counters that allow the system, after a crash, to determine which data is newer.

Atomic updates are the “Ultimate Design Problem” that all file systems have to solve.

- Solving this problem:
  - ... often involves lots of extra IO.
  - ... often is complicated and convoluted.
  - ... often is incomplete.
    - Some crashes leave the system broken, missing space, or require long FSCK operations.
- What is needed is a “better” block layer.

Introducing the ESS "Extended" Block Devices

- ESS is a software-based "block translation layer".
  - Think of it as an FTL in software that uses standard disks.
- ESS already does really neat stuff at the low level.
  - This ESS extension gives access to low-level features that just don't exist at the stock "struct bio { ... }" level.
- Not really a block device, but an "Extended Block Device" (EBD).

Feature 1: Blocks are Sparse

- Blocks have “logical block addresses” which are numbers starting at 1.
  - We avoide zero as this makes application design easier.
- There are more block numbers than there is space on the device.
  - This is convenient for many allocation schemes.
- Blocks can be available, allocated, or in-use with data.
  - The extended block device keeps track of allocations for you so you don’t have to.
- Blocks are designed to be allocated in “binary” sized groups.
  - This is similar to Linux page “buddy list” order allocations.
  - 1, 2, 4, 8, ... 32K blocks can be allocated in a single call

Feature 2: Blocks are Variable Sized

- This is where it starts to get really strange.
- With a conventional block device, all blocks are a fixed size.
- The EBD lets you store anything from zero bytes (an empty block) to 16 Megabytes as a single logical block
  - Blocks are space efficient packing data onto the media.
  - Blocks are always stored linearly without fragmentation.
  - Small and large blocks can be stored on different media types:
    - Place small blocks on Flash media.
    - Place large blocks on HDD media.

Feature 3: All Updates are Linear

- The update scheme involves 100% linear writes
  - Writes are fully compatible with “zoned” devices.
    - Zoned SSDs
    - HA/HM SMR HDDs
- Linear writes are fast
- Linear writes yield lowest flash wear
- Linear writes can be used with ultra efficient erasure code arrays.
  - The linear write logic eliminates in-place read/modify/write operations

Feature 4: All Updates are Densely Packed

- Updates, in addition to being linear are > 99% “payload”.
  - Mapping overhead is well under 1%.
  - There are no journals or double copies of data.
  - 100% of the write bandwidth is dedicated to actual data.
- Blocks can use compression.
  - … if this makes sense for your data.
  - The variable sized nature of FS control blocks makes compression less needed.

Feature 5: All updates are a part of a formal atomic update engine

- Multiple blocks can be updated together
  - The engine guarantees that either all blocks make it to disk, or none of them do.
    - ... and the previous contents persist after a crash.
- Update transactions are extensible and mergeable.
- Block allocations are part of the atomic update engine
- Atomic updates can be very long:
  - ... over a thousand blocks.
  - ... including gigabytes of data.

Enhanced Block Device Summary

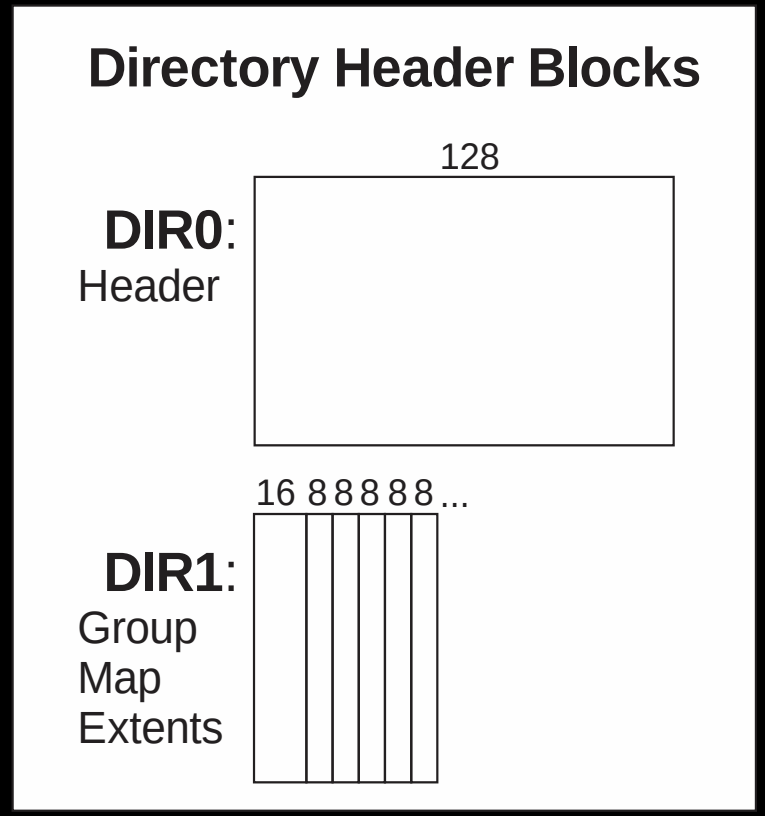
- Variable Sized Blocks
- Sparse allocation
- Atomic updates
- Linear updates at device speed
- Just what a file system needs.

The ESS for Objects File System WFFS (working title)

- This is where we start to talk about WFFS.
- WFFS sits on top of the EBD.
  - Only EBD logical addresses are used.
  - no actual disks addresses appear anywhere.
  - This lets the block layer do data moves like garbage collection without requiring FS updates.
  - This eliminates the “snowball effect” and “wandering trees” that some log structured file systems deal with.
- The file system exploits variable sized blocks to support “single IO” file access.
- The file system exploits sparse block allocation to create very compact extent tables for very large directories and files.

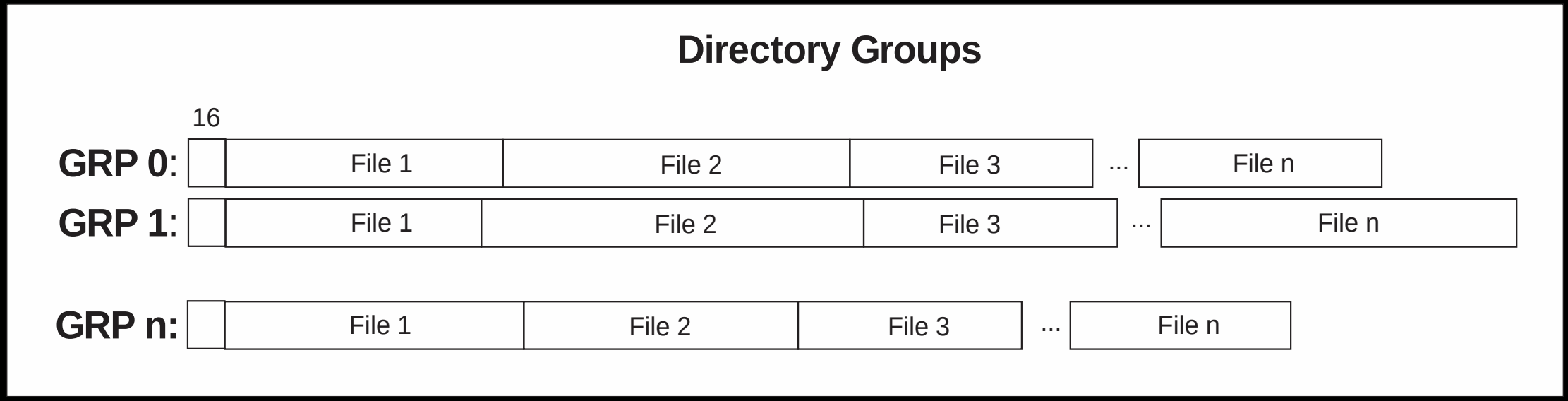
The File System Directory Structure

- Each directory has two header blocks.
- DIR0 is 128 bytes and contains counts, permissions, timestamps.
  - It gets updated frequently, so small is better
- DIR1 is variable sized and contains the group extent map.
  - Minimum is 24 bytes
  - 152 bytes supports 64K groups
  - 16 Mbytes supports 64B groups



The File System Directory Structure

- Direct Hash from the filename to a numbered group.
- Avg 32 files per group.
  - ... groups average 3-12 KB.

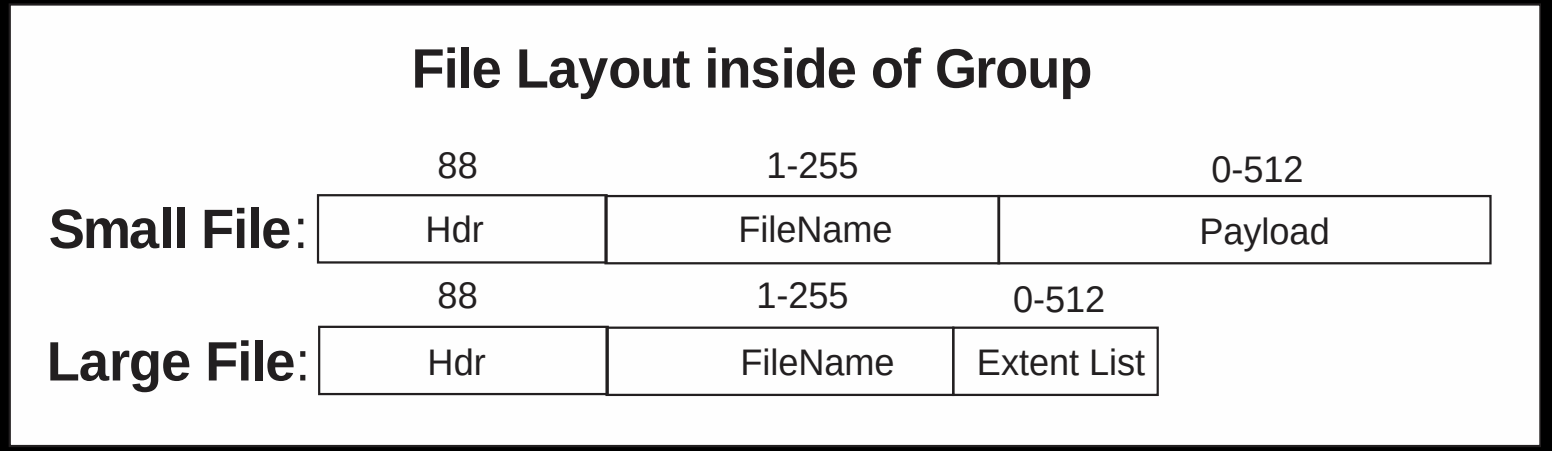


Resizable Directory Hashing

- Low cost split/merge as files are added/removed:
  - ... keeps hash balance reasonable.
- Based on a 1980s academic paper.
  - ... used in PI/Open and OpenQM.
    - a “Pick like” database you probably have never heard of.
- Ideally maps to variable sized blocks.
- Single IO to any group.

Variable Sized “File” Entries

- 1-255 byte file name / 0-512 byte payload.
- Fast search by HASH followed by filename compare.
  - Filename compare is a memcmp, as the length has already been validated by the hash.



Files have “internal” payloads

- Up to 512 bytes:
  - ... store data internally.
- Over 512 bytes:
  - ... store the extent table internally.
  - The internal extent table support files up to 250TB.

How the EBD mitigates issues with a hashed / resizable lookup scheme.

- Hashed groups will vary in size
  - Variable sized blocks directly map this
- The split/merge operations make the variability worse
  - … again, the variable sized block just work.
- Updates require consistent upgrades across several LBAs
  - The atomic update engine is ideal for this
- Both directory and file extents need to support both small and huge sets of LBAs
  - An exponential allocation scheme is used that exploits the block layers sparse LBAs
    - 512 GB files required only 128 bytes for their extent tables.
    - Directories can map 1 Billion files with a single 16 Megabyte “extent list block”.

Disk Region Layout

- This FS implementation is designed to work with a combination of “some Flash” combined with a single large HDD.
  - The HDD can be CMR or HA-SMR/HM-SMR
  - The Flash can be shared across multiple HDDs using partitions or LVM.
- There are four regions.
  - The first three are mirrored between Flash and the HDD.

Region 0:

- The first zone is used in CMR mode for some structures that are needed during mount.
  - This region is write only, except during mount.

Region 1:

- This region comprises 0.2% of the disk and is used as “virtual memory” to store the LBA “map”.
  - Most map structures cache very well, so reads from this region are rare except for individual files/extents.
  - This region is always on Flash, so reads are quite cheap.

Region 2:

- This region comprises 3-10% of the disk and is the primary storage area.
- It is also where the atomic update engine stored the authoritative copy of all data.
  - All FS structures including directories and small to medium sized file contents are stored here.

Region 3:

- This is the rest of the disk.
- It is used for large file content and is 100% data payload with no control information.
- Region 3 runs secondary to the authoritative data region 2.

Worst Case IO Analysis: File Read – Very Small Files

- ... directory is already open
- File < 512 bytes
  - 1 LBA read
  - 2 flash reads (including map)
  - 2 x flash IO latency – 0.4 ms
- XFS HDD baseline
  - 31-1000+ ms – 75X +

Worst Case IO Analysis: File Read – Small Files

- File > 512 bytes < ~256K bytes
  - 2 LBA read
  - 4 flash reads (including map)
  - 4 x flash IO latency – 0.8 ms
- XFS baseline
  - 40+ - 1000+ ms – 50X +

Worst Case IO Analysis: File Read – Large Files

- File > 256K bytes
  - 2 LBA read
  - 2 flash reads (including map)
  - 1 HDD read
  - 2 x flash IO latency + 1 x HDD IO latency – 5.4 ms
- XFS baseline
  - Small files – 10X
  - Very large files – 2X

Worst Case IO Analysis: Directory Open

- ... parent directory is already open
- 1 Small file read
- 2 LBA reads
  - reads are concurrent
  - map is contiguous
  - reads are all Flash
- 5 flash reads
- 4 x flash IO latency – 0.8 ms
- XFS Baseline
  - 50 – 1000+ ms – 60X +

IO Concurrency

- Flash IOs can run in parallel.
- with 500,000 IOPS SSD shared among 12 HDDs.
  - !40,000 IOPS per drive.
- ~ 20,000 small file reads/sec.
- ~ 10,000 medium file reads/sec (depending on file size).
- ~ 10,000 directory opens/sec.
- HDD IOs are single threaded.
  - ~100 large file reads/sec (depending on file size).
  - IO BW approaching HDD linear bandwidth.

Impact of Memory

- Map caches very effectively
  - Reduces single FILE IO latency to 0.2 ms
- Large files have limited buffering
  - Avoids cache churn where large files are read/written.

Very Early Benchmarks Performance vs XFS

- 25,000,000 files
  - Note that 25M files @ 3.5ms per file is 22+ hours
    - XFS was slowing down dramatically at large directory sizes.
  - Still some queue-depth issues reading groups in parallel.
  - Still some locking issues around directory group split/merges.

25,000,000 files		File Create	File Read
WFFS	4K	5000 / sec	1500 / sec
	128K	1400 / sec	350 / sec
XFS	4K	4000 / sec	< 30 / sec
	128K	285 / sec	< 30 / sec

Still a “Prototype”

- WFFS is far from ready for production use
  - Early performance numbers are promising and will only get better.

WILDFIRE™ STORAGE

https://WildFire-Storage.com
Doug Dumitru, CTO
doug@wildfire-storage.com
+1 949 291-0184