Storage Developer Conference
September 22-23, 2020

SDC 20
BY Developers FOR Developers

Fine Grained Block Translation

Douglas Dumitru
EasyCo LLC
www.WildFire-Storage.com
doug@easyco.com
+1 610 237-2000 x43

Legal Stuff

Some of the methods discussed here are covered by US Patents 8,380,944 , 8,812,778 , 9,092,325 , 9,535,830 and 10,248,359. Contact EasyCo LLC for licensing information.

Some of the methods discussed are licensed under open-source, non-restrictive BSD style licenses. These include the LZ4 compression algorithms and others.

Any trademarks mentioned are the property of their respective companies.

Some methods are in the public domain, such as RAID-5 and RAID-6, dynamic hashing, and others.

Some methods are released under the GPL.

The Benefits of Block Translation

- Block translation lets you change the nature of the IO workload that a device sees.
  - Convert random writes into linear writes
  - Lower the overhead and wear of parity RAID
    - … to levels that are lower than mirroring
- Block translation is also required for:
  - Data reduction like compression and de-dupe
- … But block translation can do a lot more

Block Translation

- We see frequent examples of using block translation with storage.
- How complex these techniques are largely depends on how “dynamic” the translation scheme is.

Static Layouts

- Static Layouts are common. These are layouts where the location of a block does not change and can instead be computed.
  - Partition Tables
  - RAID
  - LVM (sometimes)

Coarse Grained Dynamic Layouts

- Some layouts do move around as data changes. These solutions often involve large blocks to keep the management of the moves low cost.
  - Bad block re-mapping
  - LVM resizing
  - Copy on Write
    - LVM Snapshots

Medium Grained Dynamic Layouts

- Some solutions are in-between with a larger number of blocks that move around
  - ZFS

Fine Grained Dynamic Layouts

- Finally, some solutions are very dynamic with every block being re-positioned on every update.
  - SSD FTLs
  - Various proprietary mapping engines

Fine Grained Translation Layers are actually a Big Database Problem

- 40TB of storage is 10 Billion 4K blocks
  - Managing 10 billion of anything is non-trivial
  - Even as a simple flat table this is:
    - 34 bits x 10B = 42.5 GB
    - 5 bytes x 10B = 50 GB
    - 8 bytes x 10B = 80 GB
- The math for 400TB gets even worse

Choosing a Mapping Size

- 8 bytes entries seem ideal
  - Aligned for speed
  - Able to pack into pages
  - But the memory footprint is higher
- Bit alignment is tempting
  - Minimum memory usage
  - But you have to use a lot of shifts and locks to access entries
    - … probably not worth the trouble
- 5 bytes seems to be the sweet spot with current hardware
  - Unlike older systems, unaligned memory accesses no longer have the extreme performance hits of days past.

The Real Problem is Data Consistency Across a Crash

- We are updating two different data sets.
  - The data blocks
  - The map to the data blocks
- We cannot just update a RAM table
  - Something has to keep the two sets in sync
- The solution is “atomic updates”

Atomic Updates

- We want to write the data blocks and the control information that describes the data blocks in an "atomic update".

Atomic Update Styles

- Journals
  - not great as we duplicate the writes
- In-place pointer updates
  - Not great as this breaks any chance of linearization
- Generation counters and in-place “journal like writes”
  - This is how we will get this to work

Flash back to 2005

- I did my first work on this in 2005
- My patents date back to 2007
- The goal as:
  - Single update maintaining perfect linearity
  - Move all of the overhead to mount

The On-Disk Update Structure

- So we get to dive into the actual update structures..
- First, the update “patterns”:
  - The disk is segmented into large “write stripes”
    - The stripes are large enough to keep the media and any underlying FTL happy.
      - This is typically 256 MB for SATA SSD, and 1GB for NVMe SSDs.
      - The media does not EVER have have a write seek that is not aligned with these stripes.
  - The write stripes are further divided into “write buffers”
    - We do this so that our already large memory footprint does not go thru the sky
  - The write buffers are further divided into variable length “write segments”
    - These write segments are the “atomic update” structure

The Write Segment Structure

- At it's simplest, the structure has:
  - A header
  - Some number of data blocks
  - Some number of meta tags that describe the blocks
  - Some CRC or Hash values to verify the data is intact
  - A footer
- This is the FBD v1 layout

Atomic Update Segment Layout

HDR Meta Tag Array	Data Block	Data Block	Data Block
Data Block	Data Block	Data Block	Data Block
Data Block	Data Block	Data Block	Data Block
Data Block	Data Block	Data Block	Data Block
Data Block	Data Block	Data Block	Data Block
Data Block	Data Block	Data Block	Data Block
Data Block	Data Block	Data Block	Data Block
Data Block	…	HDR Meta Tag Array	HDR Meta Tag Array

What is in the Header?

- A signature
- The size of the segment
- The number of elements in the segment
- The generation counter for the write stripe
- A CRC or hash to ensure the segment is intact

What about Trim?

- This structure makes it very easy to implement trim/discard.
  - Leave the data block out
  - Output the meta array entry with bits that indicate all zeros or FFFFs
- Trim is amazingly fast, and has almost no wear
  - We routinely clock trim at > 500 GB/sec

This is Enough to Work Amazingly Well

- Mount is a bit slow, but safe
  - You have to ‘walk” every segment on this media
  - You don’t have to read the actual data blocks
- It is very space efficient with meta tags at only about 0.15% of space overhead
- It has size limits because of mount time
- It is not very extensible

Write Segment Optimzation #1

- Summarize the meta array for each write block.
    - This keeps you from having to walk each segment.
        - Mount is typically “two reads” for each write block
        - You still have to walk the “tail write”, but there is only one of these.
    - You do lose an additional 0.15% of space
        - Trading space for time and memory will become a theme
  - This pushes the practical array size to well over 200 TB

So Where Does This Get You?

- If you combine this write structure with an underlying optimized array you can reach:
  - Example 24 SATA SSD array running RAID-6
    - > 11 GB/sec writes at large blocks
    - > 2M 4K random writes
  - Negligible overhead on reads
  - Lowering of Flash Wear to near theoretical limits based on free space

What Does the Array Need to Do?

- The Array must be optimized for aligned writes to exact chunks
  - This is an ideal environment for erasure codes
  - Standard RAID-5/6 works very well if:
    - GPL patch for drivers/md/raid5.c with logic to avoid Read/Modify/Write cases
    - Patch also does parity calcs on calling thread, so you scale with cores.
- 11+ GB/sec and 2M+ IOPS are for 24 SATA SSD running RAID-6

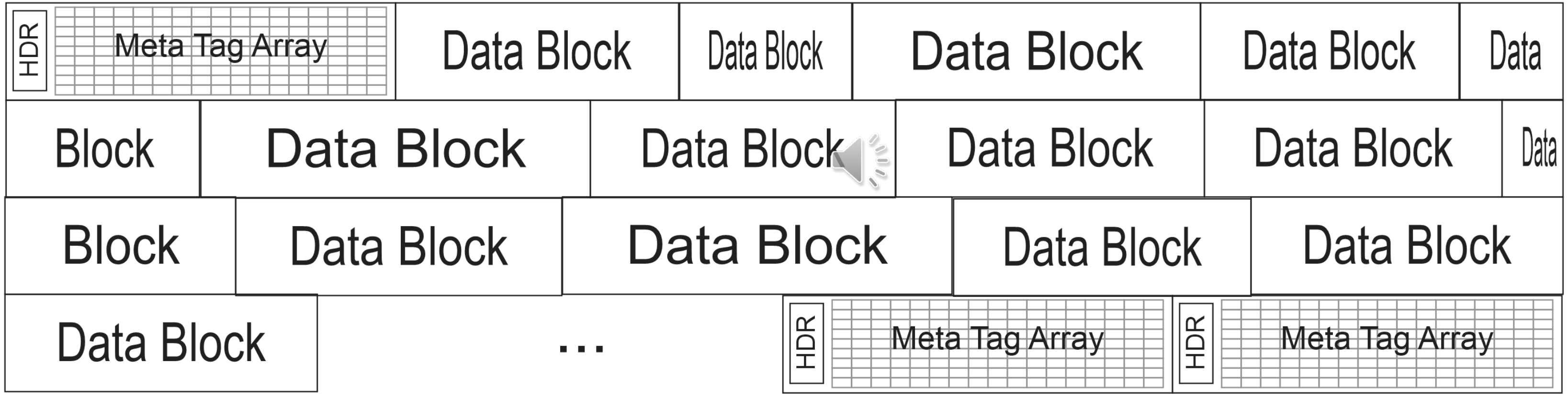
Things the Simple Layout Achives

- All data blocks are 4K aligned
  - No extra copies
- Large writes are stored together
  - Subsequent read transfers are more efficient
- Chunks are typically small, making the array scale better
  - Large reads actually spread out across multiple SSDs in parallel

But We Can Go Beyond 4K

- While 4K only blocks are efficient, the “linear write” structure lets you do more
    - Step 1, compression
        - LZ4 Optimized for 4K blocks
            - > 1GB/sec/core by limiting dict size so that it fits in L1 cache
        - Pad compressed blocks to 64 byte boundaries
            - +12 bits in the lookup table
            - 6 bytes needed for up to 256 TB arrays

Write Segement with Compressed Blocks



Compression Overhead

- Significantly more CPU usage
- Some additional latency for low Q depth IO
- But is it still easy to reach “drive speed”
  - >11 GB/sec writes
  - > 2M IOPS on writes

Beyond Just Translating Blocks

- We can abuse the atomic write segment structure to create whole new solutions
  - Step 1, lower the memory requirements and lower the amount of time for a mount
    - This is necessary for really large arrays
    - This is necessary for dual-ported drives in HA/failover configurations

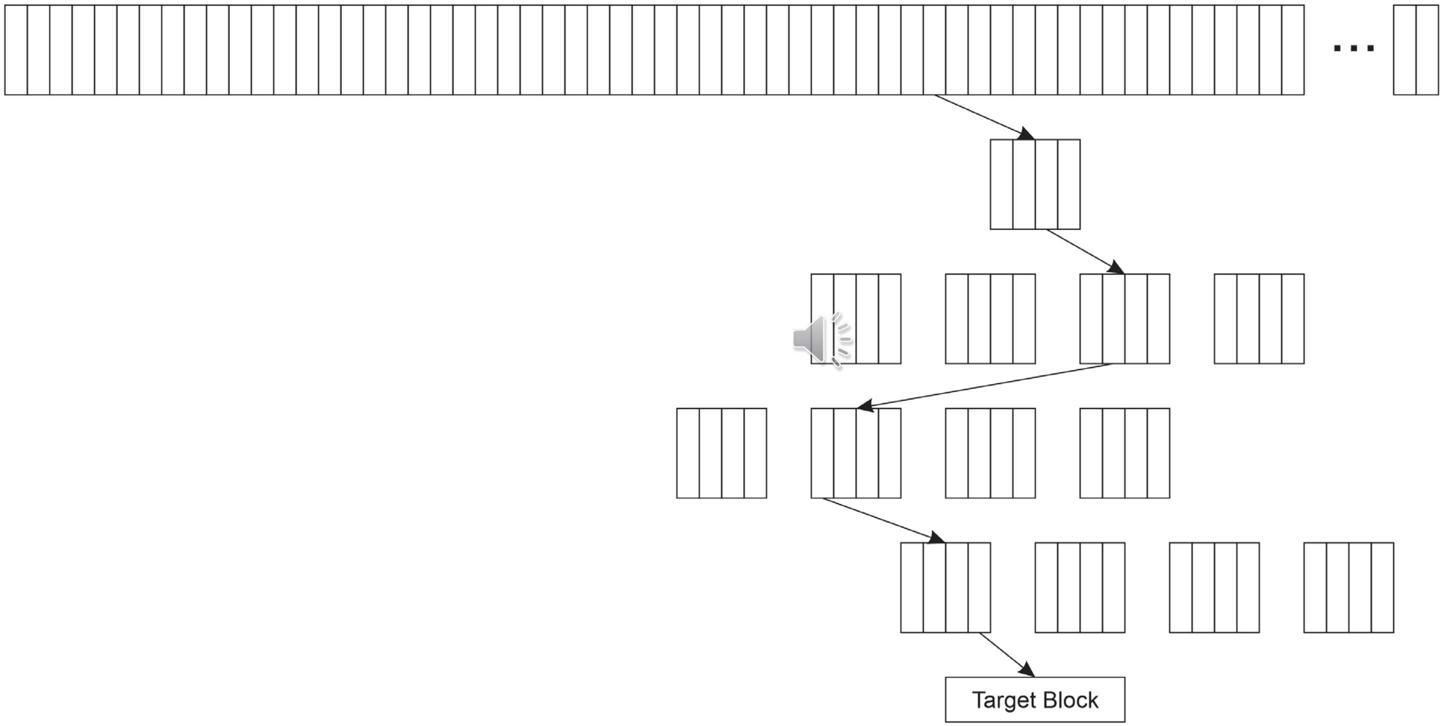
Embed Most of the LBA Table on Flash

- Place parts of the LBA table inside of the atomic write structure.
  - Each write will include a small LBA snippet that has lookups for 4, 8, or 16 LBA entries
  - This is not a "cache"
    - It is updated in real-time with the data blocks
  - Updating as few as four entries reduces the lookup table DRAM requirements by 75%

Turning the LBA Table Into a Shallow Tree

- The LBA "snippets" are very small
  - They don't add much to write overhead
  - They have little impact on Flash capacity.
- We can push this further by writing a "mini tree"
  - 2.5% of flash space overhead (worst case)
  - 1:256 DRAM usage reduction

LBA Table Lookup Tree



LBA Trees

- 40TB – we would typically need 10B entries
  - 60 GB of ram at 6 bytes/per entry
  - 1/256 of space needed – 240 MB
    - Each level is 6 bytes * 4 – 24 bytes
    - Each random write uses 96 extra bytes of storage (2.5%). Linear writes use less.
    - Nodes are easy to cache as 6 byte is enough to double as a disk or RAM pointer.
  - Only the top-level has to be mounted. The rest can be demand paged (although perhaps we need another word than "paged").

Extended Write Segment Structure

- Master Header – total sizes, counts, and generation tags
- Sub Headers, one for each type of data
    - Block data
    - Meta tags used on mount
    - Meta tags used on defrag (GC)
    - CRC arrays used for data validation
  - We still use summary meta arrays for each write block
- Mount can now read a fraction of the array
  - Enables support for “huge” arrays
  - Enables fast mounts that are quick enough for “HA Failover”

What can we use “low memory” for?

- Lowering system cost.
- De-dupe with high data reduction ratios
  - De-dupe needs:
    - LBA to Block ID translation
    - ID to actual data translation with reference counters
  - The LBA entries are most of the space, but they have high “locality” so they cache well.
  - The Block ID entries are hashes, so they need to have committed memory or else you will pay with a read on every access.

And finally, what about "beyond blocks"?

- So far, we are just storing someone else's blocks.
- We can map many LBA "numbers" to storage objects.
  - The objects can be variable sized.
  - The object can also be compressed.
  - Different types of objects can be mapped in different name spaces.
  - The LBA ranges are "thin provisioned" which makes file system design easier.
  - The LBA tables themselves are "allocation bitmaps"

So, Lets Build a File System

- This is unlike any file system you have ever seen.
  - It has no real concept of a “page”
  - It is optimized for “object access”, but still allows in-place updates.
  - Most file creates require fractional IOs.
  - Directories above 1 billion entries are practical
  - Space utilization for small files is > 90%.

The WildFire File Sysytem

- Some of this actually has been implemented
  - Ie, a single directory that can do creates, reads, and scans, and deletes.
    - Here are the “block namespaces” used so far.
      - Block type 1: Directory Control
      - Block type 2: Directory extents
      - Block type 3: Directory groups
      - Block type 4: Small File contents
      - Block type 5: File extends
      - Block type 6: Random access file blocks
      - Block type 7: Object file blocks

But What are the Block Limits?

- If you push the “block pointer” to 8 bytes, you can point to a block with:
  - Up to a 16 PB array
  - Variable sizes from 16 byte to 1 MB w/ compression
  - … or 64 MB without compression
    - Point to the local array, or off-array (perhaps to spinning media for large objects)

So What Does a Directory Look Like?

- Each directory has a single, small control block with counters.
  - This gets updated a lot. Because it is small, the overhead on disk wear and bandwidth is low.
- There is an "extents" block that maps to groups

Groups and Variable Sized Blocks are Meant for Each Other

- Groups are added and removed dynamically, but still use hashes for lookups
  - This involves hashes to a binary modulus and single group split/merges as files are created and removed
- Groups will vary in fill level, but the variable sized blocks map this perfectly.
  - Space utilization is excellent
  - … which also translates to efficient use of disk IO bandwidth

What Kind of Performance Can you Get?

- When running “from disk” (ie, nothing in cache)
  - Each file open is a single direct read
  - Each file write is a single read, and then a merge/update of the group
    - The write itself is a part of the coalesced “write stream”, so it is very low overhead.
    - Unless there is a split, then there are updates to two blocks.

What Kind of Performance Can you Get?

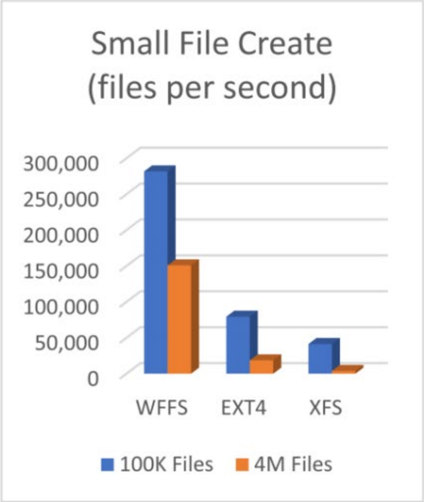
- There is “no page cache”
  - Writes go directly into the “write segment” buffer
  - The “write segment can merge multiple requests before an actual update goes out.

Small File Performance

- These were run on a Core-i7 VM with a single SATA SSD.
  - Create small files in a one directory
    - < 4 uSec per file create for 100K
    - < 7 uSec per file create for 4M
      - 2 uSec of this is VFS
  - 4x – 10x faster than EXT4
  - 8x – 30x faster than XFS
  - “Lots” faster than ZFS

File Create Benchmarks

Small File Create: 100 byte files, 20 byte filename, single thread, single SATA SSD, sync at end:



Small Files as Objects

- Small files don't really need extents
  - You can store "very little" files with their directory entries
  - You can store "bigger" files with their data in a single variable sized block
  - Optimized for files that are created "all at once"
    - But still support random and append files

Small Files as Objects

- This design is "built for speed"
  - Minimize IO with direct access to the data
  - Keep the data structure tight
    - All direct RAM links
    - Count "cache line misses"
    - No BTREEs or other slow lookups
    - No allocation bitmaps
    - All data is written "exactly once"
      - No journals

Really Big Directories

- 1 billion item directories are practical
  - It is easy to sustain creates at 2500/sec, single thread
  - Under 5 days for 1B file creates
    - vs 20 months for EXT4 and 6 years for XFS
    - … and even longer (actually much longer) for ZFS
- While everyone talks about file systems “without limits”, the limit that matters most is “time”

Incredible Space Efficiency

- 1 billion 100 byte files with 20 byte file names
  - About 200GB of space used with WFFS
  - About 5 TB of space used with EXT4/XFS
- … and it all works because you are translating block addresses.

But Why Is Block Translation Needed for a File System?

- Block Translation keeps the design simple
- Space management can happen in the background
  - This allows for defrag (GC)
- Thin provisioning of block addresses lets you assign large linear ranges when needed or individual blocks when needed.
- The atomic update structure is not an “extra write”.
- Writes are nearly 100% actual data with no pad.

Fine Grained Block Translation

Douglas Dumitru
EasyCo LLC
www.WildFire-Storage.com
doug@easyco.com
+1 610 237-2000 x43

Please take a moment to rate this session.

Your feedback matters.