BTRFS Declustered Parity RAID For Zoned Devices

14 September, 2022
<table>
<thead>
<tr>
<th>Outline</th>
<th>Proposed Changes</th>
<th>Design Background</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>• Btrfs Overview</td>
<td>• Distributed Data Placement</td>
</tr>
<tr>
<td></td>
<td>• Zoned Devices</td>
<td>• RAID Stripe Tree</td>
</tr>
<tr>
<td></td>
<td>• ZONE APPEND Write Operations</td>
<td>• Current Status</td>
</tr>
<tr>
<td></td>
<td>• Btrfs On Zoned Devices</td>
<td>• Outlook</td>
</tr>
<tr>
<td></td>
<td>• Problem Statement</td>
<td>• Screenshots</td>
</tr>
<tr>
<td></td>
<td>• Lessons Learned From RAID5/6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Background
Btrfs Overview

What’s btrfs?

- Copy-on-Write Filesystem
  - Based on CoW B-Trees
  - Snapshots
  - Subvolumes

- Additional Features
  - Transparent data compression
    - lzo, zlib or zstd
  - Checksums for data and metadata
    - crc32c, xxhash64, sha256, blake2b

- Built-in multi device support (RAID)
  - RAID 0, RAID 1, RAID 10, RAID 5, RAID 6

- Incremental backups with send/receive
  - Send stream of changes between two subvolume snapshots
Zoned Block Devices
What’s ZBC, ZAC And ZNS?

• Most commonly found today in the form of SMR hard-disks (Shingled Magnetic Recording) or ZNS SSDs
  • Defined in SCSI ZBC, ATA ZAC and NVMe ZNS

• LBA range divided into zones
  • Conventional zones
    • Accept random writes
  • Sequential write required zones
    • Writes must be issued sequentially starting from the “write pointer”
    • Zones must be reset before rewriting

• Users of zoned devices must be aware of the sequential write rule
  • Device fails write command not starting at the zone write pointer
ZONE APPEND Write Operations
Introduced with NVMe Zoned Namespace (ZNS) SSDs

• ZONE APPEND write operation only specifies the target zone
  • The device automatically write at the current write pointer position of the zone
  • The first written LBA number is returned to the host with the command completion notification

• ZONE APPEND command is not defined in the ZBC (SCSI) and ZAC (ATA) standards
  • Emulated in the SCSI disk driver since kernel version 5.8
  • With zone append, writes to a zone can be delivered in any order without failing
  • User must however be ready to handle out-of-order completions
<table>
<thead>
<tr>
<th>What we’ve done</th>
<th>Fully functional since kernel v5.12</th>
<th>NVMe ZNS support since kernel v5.16</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Basic support merged with kernel v5.11</td>
<td>- Use ZONE APPEND for data writes</td>
<td>- Zone capacity smaller than zone size</td>
</tr>
<tr>
<td>- Log structured super block</td>
<td>- Not yet completely on par with regular BTRFS features</td>
<td>- Respecting queue_max_active_zones() limits</td>
</tr>
<tr>
<td>- Superblock is the only fixed location data structure in btrfs</td>
<td>- No NOCOW</td>
<td>- Currently in stabilization phase</td>
</tr>
<tr>
<td>- Align block groups to zones</td>
<td>- No fallocate(2)</td>
<td>- Automatic zone reclaim merged in v5.13</td>
</tr>
<tr>
<td>- Zoned extent allocator</td>
<td>- No RAID yet</td>
<td>- Bug fixes for corner cases</td>
</tr>
<tr>
<td>- Append only allocation to avoid random writes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem Statement
Problem Statement
Lessons Learned From Btrfs RAID5/6

• Disconnection of "File-Extent-Layer" and "RAID-Layer"
  • Sub stripe length updates in place
    • RAID Write Hole
    • Not possible on a zoned btrfs
  • CoW needs to know about RAID and vice versa
  • Needs to work with “nocow” files/filesystem as well
Problem Statement
Lessons Learned From Btrfs RAID

• Implicit data placement
  • Each per disk sub-stripe has same offset from chunk start

• Doesn’t work with a zoned filesystem (even for RAID 1)
  • Multiple writes to different drives can race
    • No explicit write position with zone append command: the drives decides

Deterministic Placement vs. Non-deterministic Placement
Problem Statement
Lessons Learned From RAID

- RAID Rebuild Stress
  - RAID5 can only tolerate one missing drive, two for RAID 6
  - High stress on remaining drives for rebuild
  - Increased chance of disk dying during rebuild

- Inflexible Encoding Scheme
  - XOR for RAID 5 (P-Stripe)
  - XOR and Shift for RAID 6 (Q-Stripe)
Proposed Changes
Proposed Changes
How to fix these problems

- Distribute Data Placement
  - Similar to what BTRFS RAID 1 already does
  - Less pressure on single disks in recovery

- Copy-on-Write to circumvent write hole
  - Introduce RAID Stripe Tree
  - Write data first, then meta-data describing the stripe
  - Allows us to use REQ_OP_ZONE_APPEND for zoned data writes

- Configurable Parity Algorithm
  - None (RAID 0/1)
  - XOR/P-Q Stripe (RAID 5/6)
  - Erasure-Codes: Reed Solomon or MDS Codes (more than 2 blocks of parity)
Design Background
Design Background

Distributed Data Placement

- Traditional RAID6 (2D+2P)
- Dataset + parity is striped across all disks

![Diagram of RAID 6 volume (2D+2P)]
Design Background

Distributed Data Placement

- Traditional RAID6 (2D+2P)
  - Dataset + parity is striped across all disks

- Declustered RAID (2D+2P)
  - Dataset + parity is distributed among a subset of disks
Design Background
RAID Stripe Tree

- Can be seen as an inverse of the free space tree
  - Written after the data has reached the disks
  - Records the location (disk, LBA) of each sub-stripe
- Kind of RAID “journal”
  - Removes write hole (CoW)
  - Can be use for “nocow” as well
- Logical to physical addresses translation
  - Logical (start, length) tuple maps to N (disk, start) tuples
**Design Background**

**RAID Stripe Tree**

- Translate logical to physical addresses (3D + 2P)

Logical Space

File

File Extent 0-3M

File Extent 24M-27M

Physical Space

- Device
  - Stripe Extent 0-1M
  - Disk 0
  - 256M + 1M

- Device
  - Stripe Extent 1-2M
  - Disk 4
  - 128M + 1M

- Device
  - Stripe Extent 2-3M
  - Disk 3
  - 1024M + 1M

- Device
  - Stripe Extent P-Parity
  - Disk 6
  - 512M + 1M

- Device
  - Stripe Extent Q-Parity
  - Disk 7
  - 2048M + 1M

Block Group
Design Background

RAID Stripe Tree

- Keyed by logical, length
- Additional per file extent space consumption
- N*16 Bytes
- Example 3D + 2P RAID
  - 5 * 16 Bytes = 80 Bytes stripe tree nodes
  - 51 Nodes per 4k sector

```c
struct btrfs_key {
    .objectid = file_extent_logical,
    .type = BTRFS_RAID_STRIPE_EXTENT,
    .offset = file_extent_length,
};

struct btrfs_dp_stripe {
    /* array of RAID stripe extents this stripe is comprised of */
    struct btrfs_stripe_extent extents[];
} __attribute__((__packed__));

struct btrfs_stripe_extent {
    /* btrfs device-id this raid extent lives on */
    __le64 devid;
    /* physical start address on the device */
    __le64 physical;
} __attribute__((__packed__));
```
Design Background

RAID Stripe Tree

```c
struct btrfs_file_extent_item {
    __le64 generation;
    __le64 ram_bytes;
    __u8 compression;
    __u8 encryption;
    __le16 other_encoding;
    __u8 type;
    __le64 disk_bytenr;
    __le64 disk_num_bytes;
    __le64 offset;
    __le64 num_bytes;
} __attribute__((__packed__));

struct btrfs_key {
    .objectid = file_extent_logical,
    .type = BTRFS_RAID_STRIPE_EXTENT,
    .offset = file_extent_length,
};

struct btrfs_dp_stripe {
    /* array of RAID stripe extents this stripe is
    * comprised of
    */
    struct btrfs_stripe_extent extents[];
} __attribute__((__packed__));

struct btrfs_stripe_extent {
    /* btrfs device-id this raid extent lives on */
    __le64 devid;
    /* physical start address on the device */
    __le64 physical;
} __attribute__((__packed__));
```
Design Background
RAID Stripe Tree

struct btrfs_file_extent_item {
    __le64 generation;
    __le64 ram_bytes;
    __u8 compression;
    __u8 encryption;
    __le16 other_encoding;
    __u8 type;
    __le64 disk_bytenr;
    __le64 disk_num_bytes;
    __le64 offset;
    __le64 num_bytes;
} __attribute__((__packed__));

struct btrfs_key {
    .objectid = file_extent_logical,
    .type = BTRFS_RAID_STRIPE_EXTENT,
    .offset = file_extent_length,
};

struct btrfs_dp_stripe {
    /* array of RAID stripe extents this stripe is
     * comprised of
     */
    struct btrfs_stripe_extent extents[];
} __attribute__((__packed__));

struct btrfs_stripe_extent {
    /* btrfs device-id this raid extent lives on */
    __le64 devid;
    /* physical start address on the device */
    __le64 physical;
} __attribute__((__packed__));
Design Background

RAID Stripe Tree

```c
struct btrfs_file_extent_item {
    __le64 generation;
    __le64 ram_bytes;
    __u8 compression;
    __u8 encryption;
    __le16 other_encoding;
    __u8 type;
    __le64 disk_bytenr;
    __le64 disk_num_bytes;
    __le64 offset;
    __le64 num_bytes;
} __attribute__ ((__packed__));
```

```c
struct btrfs_key {
    .objectid = file_extent_logical,
    .type = BTRFS_RAID_STRIPE_EXTENT,
    .offset = file_extent_length,
};
```

```c
struct btrfs_dp_stripe {
    /* array of RAID stripe extents this stripe is 
     * comprised of 
     */
    struct btrfs_stripe_extent extents[];
} __attribute__ ((__packed__));
```

```c
struct btrfs_stripe_extent {
    /* btrfs device-id this raid extent lives on */
    __le64 devid;
    /* physical start address on the device */
    __le64 physical;
} __attribute__ ((__packed__));
```
Design Background
RAID Stripe Tree

```c
struct btrfs_file_extent_item {
    __le64 generation;
    __le64 ram_bytes;
    __u8 compression;
    __u8 encryption;
    __le16 other_encoding;
    __u8 type;
    __le64 disk_bytenr;
    __le64 disk_num_bytes;
    __le64 offset;
    __le64 num_bytes;
} __attribute__ ((__packed__));
```

```c
struct btrfs_key {
    .objectid = file_extent_logical,
    .type = BTRFS_RAID_STRIPE_EXTENT,
    .offset = file_extent_length,
};
```

```c
struct btrfs_dp_stripe {
    /* array of RAID stripe extents this stripe is
     * comprised of
     */
    struct btrfs_stripe_extent extents[];
} __attribute__ ((__packed__));
```

```c
struct btrfs_stripe_extent {
    /* btrfs device-id this raid extent lives on */
    __le64 devid;
    /* physical start address on the device */
    __le64 physical;
} __attribute__ ((__packed__));
```
# Design Background

## RAID Stripe Tree

### Advantages

- Address translation
- Scrub friendly
- RAID Journal
  - Ordered updates
  - Similar to how checksums are handled

### Advantages

- No implicit connection needed
- `REQ_OP_ZONE_APPEND` compatible
- Stronger reliability against device faults
  - `M+K` erasure code can be high
Design Background

RAID Stripe Tree

Disadvantages

• Additional Metadata

• Especially if we also must do stripe tree entries for metadata

• Merge consecutive and sequential on-disk stripe extents?
Design Background
Configurable Parity Algorithm

- Generates Parity or EC information
- Like how we handle compression
  - Do the math on data read/write
- But different to how we handle compression
  - Doesn’t modify the actual data but adds data
Current Status
Current Status
Where are we at the moment?

- Data RAID1 and RAID1 implemented
- Metadata doesn’t use `REQ_OP_ZONE_APPEND`
- Already working out-of-the-box
- Data writes are recorded in raid-stripe-tree
Current Status

- Boilerplate mkfs creating an FS with empty RAID stripe tree

```bash
> mkfs.ext4 -d raid-stripe-tree -o raid1 -m raid1 /dev/nullb0 /dev/nullb1
> mkfs.ext4 -v 5.16.1
See https://gitlab.com/master.git for more information.

Zoned: /dev/nullb0: host-managed device detected, setting zoned feature
Resetting device zones /dev/nullb0 (100 zones) ...

NOTE: several default settings have changed in version 5.15, please make sure
- to do not affect your deployments:
  - DD for retdata (=-m dup)
  - enabled no-holes (=-o no-hole)
  - enabled free-space-tree (=-O free-space-tree)

Resetting device zones /dev/nullb1 (100 zones) ...
Label:  (null)
UUID:  56a552e0-7a87-4933-9e4a-a21f69f9132
Node size:  16384
Sector size:  512
Filesystem size:  35.8GB

Block group profiles:

- Data:  RAID1  128.00MB
- Metadata: RAID1  128.00MB
- System:  RAID1  128.00MB

550 detected:  yes
Zoned device:  yes
Zone size:  128.00MB

Incomp features:  ext2, ext3, ext4, slim, fat, no-data, no-holes, zoned
Runtime features:  free-space-tree, raid-stripe-tree
Checksum:  crc32c
Number of devices:  2

Device:
  ID  SIZE PATH
  1  12.8GB / dev/nullb0
  2  12.8GB / dev/nullb1

[ 7a.232886] BTRFS: device fsid 56a552e0-7a87-4933-9e4a-a21f69f9132 devid 1 transid 7 (dev/nullb0 scanned)
[ 7a.232886] BTRFS: device fsid 56a552e0-7a87-4933-9e4a-a21f69f9132 devid 2 transid 7 (dev/nullb1 scanned)
```

```bash
raid0l: 14 mkfs.btrfs -d raid-stripe-tree -o raid1 -m raid1 /dev/nullb0 /dev/nullb1
```
Current Status

• Tree-dump (on RAID1)
Current Status

- Fsck On An Non-Empty RAID Filesystem

```
rapido1:# btrfs check --check-data-csum --progress /dev/nullb0
Opening filesystem to check...
Checking filesystem on /dev/nullb0
UUID: 0a922716-004-4772-904c-5e90db334e2
  (1/7) checking root items                           (0:00:00 elapsed, 26 items checked)
  (2/7) checking extents                              (0:00:00 elapsed, 22 items checked)
  (3/7) checking free space tree                      (0:00:00 elapsed, 5 items checked)
  (4/7) checking fs roots                             (0:00:00 elapsed, 2 items checked)
  (5/7) checking csms against data                    (0:00:00 elapsed, 2 items checked)
  (6/7) checking root refs                            (0:00:00 elapsed, 3 items checked)
  (7/7) checking quota groups skipped (not enabled on this FS)
found 1246184 bytes used, no error found
  total csms bytes: 1924
  total tree bytes: 196648
  total fs tree bytes: 32768
  total extent tree bytes: 16384
  btree space waste bytes: 166678
  file data blocks allocated: 1048576
  referenced 1048576
rapido1:#
```