whamcloud

Lustre User Group
Orlando Fl
April 2011

Btrfs: Overview & Requirements for a btrfs-osd

- Johann Lombardi

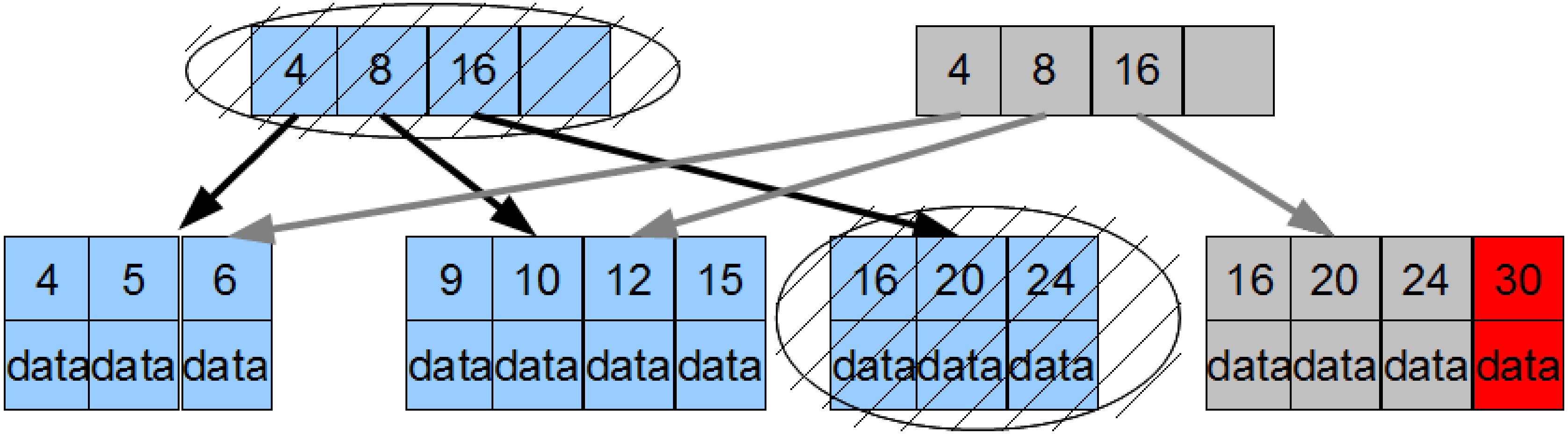
Principal Engineer
Whamcloud, Inc.

Agenda

- Quick Dive into Btrfs Internals
- Some Cool Btrfs Features
- Btrfs as Backend Filesystem for Lustre

Btrfs Btree

- Stores key/item pairs
- B+tree w/o linked leaves
- Modified in a Copy-On-Write (COW) manner
- Reference counting for filesystem trees
- Everything in the filesystem is an item of the COW btree
  - inodes, directory entries, file data, checksums, …
- Collections of btrees
  - extent tree, subvol trees, chunk tree, …



Btrfs Key

```
struct btrfs_disk_key {
    __le64 objectid;
    u8     type;
    __le64 offset;
}
```

- Objectid (8 bytes)
  - Each logical object (inode) has an unique id
  - This id is reported as the inode number
  - Most significant bit of the key
  - All items of a given objectid are grouped together in the btree

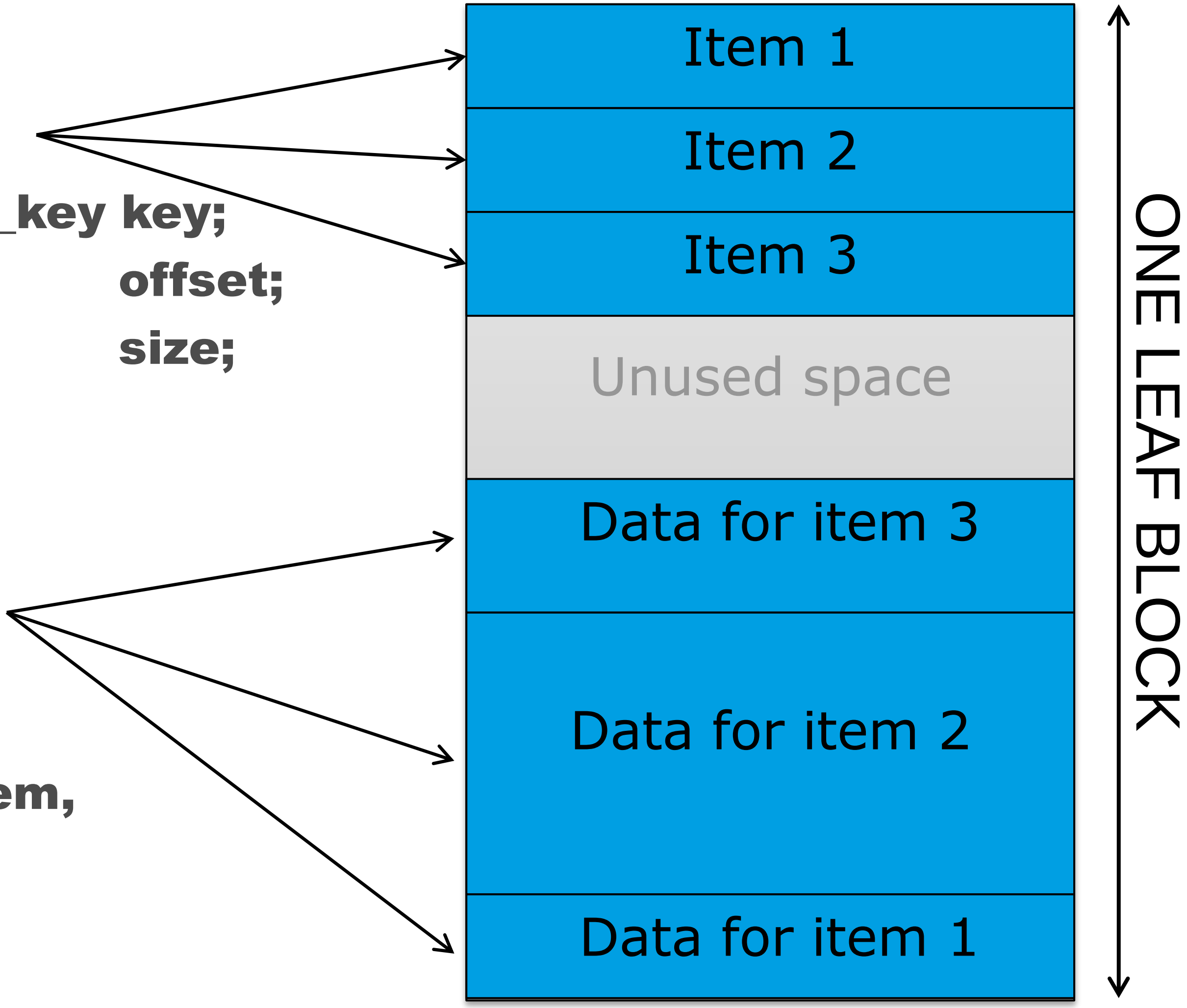
- Offset (8 bytes)
  - Offset for a particular item in the object
  - For file extent, byte offset of the start of the extent inside the file
  - But also used to store hash for directory entry lookup

- Type (1 byte)
  - Item type
  - e.g. inode, directory entries, xattr, extent, ….

Btrfs Item

```
struct btrfs_item {
    struct btrfs_disk_key key;
    __le32                 offset;
    __le32                 size;
}
```

btrfs_inode_item,
btrfs_dir_item,
btrfs_root_item,
btrfs_file_extent_item,
…



Directory Structures

- Double indexation: by name & by inode number
- 1st index for filename lookup

directory objectid	BTRFS_DIR_ITEM_KEY	64-bit filename hash

- 2nd index for readdir to return data in inode number order

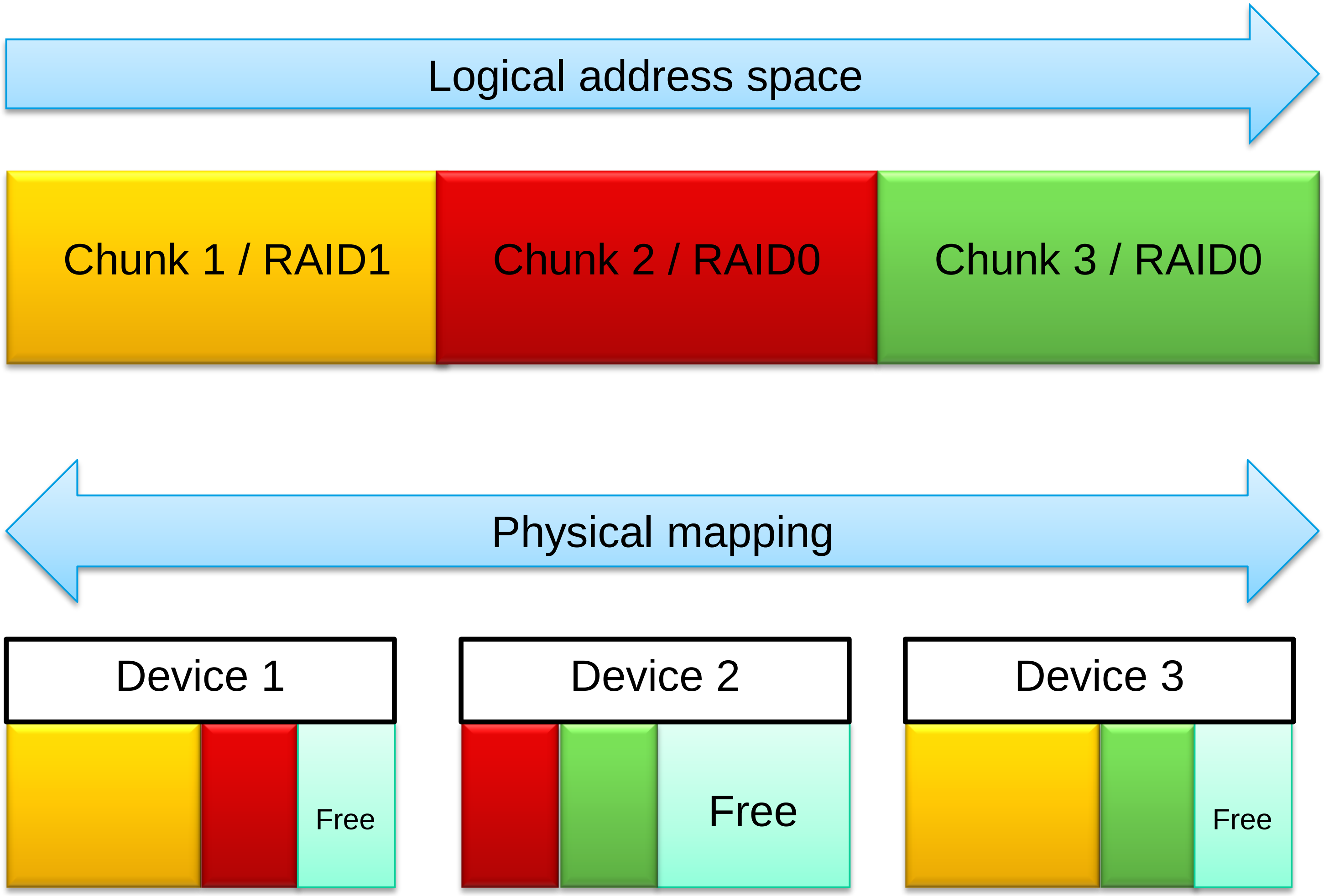
directory objectid	BTRFS_DIR_INDEX_KEY	inode sequence #

- Back reference from inode to parent directory
- Metadata overhead quite larger than for ext4
  - 3 items required for a link and each one stores the filename

Storage Pool Management

- Create pool of storage out of all devices
- Allocate space for filesystem’s use in chunks (1GB+)
- Put all those chunks together to create a logical address space
  - Different type of groups: system, metadata, data
  - Different RAID configurations (RAID 0/1/10)
  - Logical addressing allows efficient chunk relocation
- Logical to physical address mapping handled by a dedicated btree, namely the **chunk tree**

Chunk Tree



Logical address space

Chunk 1 / RAID1

Chunk 2 / RAID0

Chunk 3 / RAID0

Physical mapping

Device 1

Free

Device 2

Free

Device 3

Free

Some Cool Btrfs Features (1/2)

- List recently modified files **very** quickly
  - No need to scan every single inode as with e2scan
  - Parse btree and use generation pointers to identify parts of the btree that have been modified since a given transid

```
$ btrfs subvol find-new / 38276
inode 188677 file offset 1662976 len 4096 disk start 18154328064
offset 0 gen 38279 flags NONE var/log/kern.log
inode 188678 file offset 307200 len 4096 disk start 18154680320
offset 0 gen 38280 flags NONE var/log/auth.logtransid
marker was 38277
```

Some Cool Btrfs Features (2/2)

- Store small files (≤4k) in the btree leaves
  - Max size configurable through max_inline mount option
- Checksum support
  - Currently uses crc32c for data & metadata
  - Data checksums stored in a dedicated btree
- In-place conversion from ext3/ext4
  - Create a btrfs filesystem inside the free space of the ext4 fs
  - The new btrfs filesystem duplicates the metadata and points to the data blocks of the ext4 fs
  - Preserve original ext4 fs (data & metadata) as a snapshot
  - Can then choose to rollback to ext4 or delete the snapshot

Shortcomings

- Offline btrfsck still under development
- No background scrubbing yet
  - Patches available but no landed yet
- No hybrid storage support yet (like L2ARC)
  - Work underway to have allocation profiles (e.g. put metadata & log tree on SSD)
- No quota support
  - Not even space accounting
- RAID 5/6 support still under development
  - Only support RAID 0/1/10
- Lack of proper error handling
  - Still too many « ret = func(); BUG_ON(ret); » in the code

Btrfs as Backend Filesystem for Lustre

- Btrfs looks like the ideal backend filesystem for Lustre:
  - All the features of a modern filesystem
  - Already included into the kernel mainline
  - Expected to be the de facto filesystem of all Linux distributions soon
- Btrfs less mature than ZFS
- But btrfs is catching up very quickly
  - Many companies (Fujitsu, Red Hat, Intel, Novell, …) dedicate developers

Object Index

- Lustre FID to btrfs objectid mapping
- dmu-osd uses a dedicated ZAP
- ldiskfs-osd uses an IAM directory (namely oi.16)
- btrfs-osd could use a regular directory, but:
  - Metadata overhead (3 items) bigger than with other filesystems
    - 2 dir item = 2 * (25 + 30) = 110 bytes
    - 1 inode backref = 25 + 10 = 35 bytes
    - = 145 bytes = 145MB for 1M files. Maybe not such a big concern
  - Would need to increase nlink not to confuse btrfsck
    - Problem on the MDT since we return nlink to clients
- Add a new item type
  - Less overhead, but require changes to btrfsck to support the new item

Space Reservation & Lustre Grant

- Metadata overhead must be accounted to prevent ENOSPC error
- Btrfs books 96KB per item

Operation	#items	Space
creat/mknod/mkdir/link	5	480KB
unlink	10	~1MB
rename	20	~1.9MB
1MB write (no split)	1 + csum	224KB
1MB write (with splits)	(#splits + 1) + csum	(#splits + 1) * 96KB + 128KB

- Chunk allocation might also be needed
- Working on estimating the worst case scenario (max #splits)
- Lustre grant must be changed accordingly

Other Problems to Consider

- Extended Attributes
  - Striping info (LOV EA), metadata attributes (LMA), filter fid EA
  - Btrfs stores EA in a separate item
    - One EA is currently limited to the size of a leaf (i.e. 4KB)
    - Problem for large striping support
- Inode versioning
  - Needed for VBR
  - Can use the « sequence » field of btrfs_inode_item
- Write ahead log breaks transaction ordering
  - Out-of-order transactions not supported by Lustre
- No commit callback mechanism
- No btrfs functions are exported

Thank You

- Johann Lombardi

  Principle Engineer

  Whamcloud, Inc.

btrfs_header

- Each btree block has a btrfs_header
- This header includes:
  - The block number where the block is supposed to live
  - A generation number
- Everything that points to a btree block also stores the generation field it expects that block to have
- This allows to handle phantom & misplaced writes

Back Reference

- Not only from inode to parent directory
- Also file extent backrefs, btree extent backrefs, …
- Purposes:
  - Integrity check: check that a reference is valid
  - Quickly find holder of an extent, useful when
    - A given block is corrupted
    - The filesystem has to be resized (shrunk)

Allocation Algorithms

- Delayed allocation support
- Store btree of free extents on disk
- Build red-black tree to track free space in memory
- Different allocation policy for rotating media & SSD
- Different allocation policy for data & metadata

Write Ahead Logging Tree

- COW btree efficient for long running transactions
  - Commit every 30s by default (vs 5s with jbd2/ext4)
- Slow for frequent commits
- Specialized log for synchronous operations
  - e.g. fsync & O_SYNC writes
  - File or directory items copied to a dedicated btree
  - Synchronous operation on a given file only writes metadata for that one file
- Very similar to ZFS' approach with ZIL

Checksums

- Currently use crc32c for data & metadata
- Data checksums stored in a dedicated btree
- Disk format has room for
  - 256-bit checksum for metadata (= btree checksum)
  - Up to a full leaf block (i.e. 4KB) for data blocks
  - Inline data covered by checksum of the btree block

Some Other Cool Btrfs Features

- Writeable snapshots / subvolume support
- File cloning (cp --reflink)
- Transparent compression using zlib or lzo
- Online resize & defragmentation
  - Online device addition/removal
  - Online space rebalancing
- SSD optimizations
  - Trim support (-o discard)
  - Allocation optimizations