BTRFS: The Linux B-Tree Filesystem

OHAD RODEH, IBM
JOSEF BACIK and CHRIS MASON, FusionIO

BTRFS is a Linux filesystem that has been adopted as the default filesystem in some popular versions of Linux. It is based on copy-on-write, allowing for efficient snapshots and clones. It uses B-trees as its main on-disk data structure. The design goal is to work well for many use cases and workloads. To this end, much effort has been directed to maintaining even performance as the filesystem ages, rather than trying to support a particular narrow benchmark use-case.

Linux filesystems are installed on smartphones as well as enterprise servers. This entails challenges on many different fronts.

— *Scalability*. The filesystem must scale in many dimensions: disk space, memory, and CPUs.
— *Data integrity*. Losing data is not an option, and much effort is expended to safeguard the content. This includes checksums, metadata duplication, and RAID support built into the filesystem.
— *Disk diversity*. The system should work well with SSDs and hard disks. It is also expected to be able to use an array of different sized disks, which poses challenges to the RAID and striping mechanisms.

This article describes the core ideas, data structures, and algorithms of this filesystem. It sheds light on the challenges posed by defragmentation in the presence of snapshots, and the tradeoffs required to maintain even performance in the face of a wide spectrum of workloads.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: File-system management; D.4.8 [**Operating Systems**]: Performance

General Terms: Algorithms, Performance

Additional Key Words and Phrases: B-trees, concurrency, copy-on-write, filesystem, RAID, shadowing, snapshots

ACM Reference Format:
Rodeh, O., Bacik, J., and Mason, C. 2013. BTRFS: The Linux B-tree filesystem. *ACM Trans. Storage* 9, 3, Article 9 (August 2013), 32 pages.
DOI: http://dx.doi.org/10.1145/2501620.2501623

1. INTRODUCTION

BTRFS is an open source filesystem that has seen extensive development since its inception in 2007. It is jointly developed by Fujitsu™, Fusion-IO™, Intel™, Oracle™, Red Hat™, Strato™, SUSE™, and many others. It is slated to become the next major Linux filesystem. Its main features are:

(1) CRCs maintained for all metadata and data;
(2) efficient writeable snapshots, clones as first class citizens;
(3) multidevice support;

Author's address: O. Rodeh; email: orodeh@us.ibm.com.

DOI: http://dx.doi.org/10.1145/2501620.2501623

(4) online resize and defragmentation;
(5) compression;
(6) efficient storage for small files;
(7) SSD optimizations and TRIM support.

The design goal is to work well for a wide variety of workloads, and to maintain performance as the filesystem ages. This is in contrast to storage systems aimed at a particular narrow use-case. BTRFS is intended to be used by mainstream Linux distributions; it is expected to work well on systems as small as a smartphone, and as large as an enterprise production server. As such, it must work well on a wide range of hardware.

The filesystem on disk layout is a forest of B-trees, with copy-on-write (COW) as the update method. Disk blocks are managed in extents, with checksumming for integrity, and reference counting for space reclamation. BTRFS is unique among filesystems in its use of COW friendly B-trees [Rodeh 2008] and reference counting.

Filesystem performance relies on the availability of long contiguous extents. However, as the system ages, space becomes increasingly fragmented, requiring online defragmentation. Due to snapshots, disk extents are potentially pointed to by multiple filesystem volumes. This make defragmentation challenging because: (1) extents can only be moved after all source pointers are updated and (2) file contiguity is desirable for all snapshots.

To make good use of modern CPUs, good concurrency is important. However, with copy-on-write this is difficult, because all updates ripple up to the root of the filesystem.

This article shows how BTRFS addresses these challenges, and achieves good performance. Compared with conventional filesystems that update files in place, the main workload effect is to make writes more sequential, and reads more random.

It is important to understand that BTRFS is a work in progress. Features, performance improvements, and bug fixes are contributed with every minor kernel release. While we believe that it holds great promise, we also think that a lot remains to be accomplished. There has been a lot of work on the RAID subsystem, which now supports mirroring, striping, and mirroring+striping (RAID levels $\{0, 1, 10\}$). Code to support RAID-5/6 has just recently been added. Fragmentation is dealt with in a number of ways, and we are not yet ready to say that the optimal solution has been found. Flash optimization is an ongoing project, and we report on some of this work here, however we cannot claim that our system is the optimal flash storage solution.

The approach taken in this article is to explain the core concepts and intuitions through examples and diagrams. The reader interested in finer grain details can find the filesystem code publicly available from the Linux kernel archives, and low level discussions in the kernel mailing list [BTRFS].

This article is structured as follows. Section 2 describes related filesystems. Section 3 describes basic terminology, presents the B-trees used to hold metadata, and shows the fundamentals of copy-on-write updates. Section 4 is about the use of multiple devices, striping, mirroring, and RAID. Section 5 describes defragmentation, which is important for maintaining even filesystem performance. Section 6 talks about performance, and Section 7 summarizes.

2. RELATED WORK

On Linux, there are three popular filesystems, Ext4 [Mathur et al. 2007], XFS [Hellwig 2009; Sweeney et al. 1996], and BTRFS [Mason 2007]. In the class of copy-on-write filesystems, two important contemporary systems are ZFS [Bonwick and Moore; Henson et al. 2003], and WAFL [Edwards et al. 2008; Hitz et al. 1994]. In what

follows, we use the term *overwrite based filesystem* to refer to systems that update files in place. At the time of writing, this is the prevalent architectural choice.

BTRFS development started in 2007, by Mason. He combined ideas from ReiserFS [Reiser 2001], with COW friendly B-trees suggested by O. Rodeh [Rodeh 2008], to create a new Linux filesystem. Today, this project has many contributers, some of them from commercial companies, and it is on its way to becoming the default Linux filesystem. As development started in 2007, BTRFS is less mature and stable than others listed here.

The *Fourth Extended Filesystem* (EXT4) [Mathur et al. 2007] is a mostly backward compatible extension to the previous general purpose Linux filesystem, Ext3. It was created to address filesystem and file size limitations, and to improve performance. Initially, Linux kernel developers improved and modified Ext3 itself, however, in 2006, Ext4 was forked in order to segregate development and changes in an experimental branch. Today, Ext4 is the default Linux filesystem for some common Linux distributions. As it is an in-place replacement for Ext3, older filesystems can seamlessly be upgraded. Ext4 is an overwrite-based filesystem, which manages storage in extents. It uses an efficient tree-based index to represent files and directories. A write-ahead journal is used to ensure operation atomicity. Checksumming is performed on the journal, but not on user data. Snapshots are not supported internally, rather, the underlying volume manager provides that functionality.

XFS [Hellwig 2009] is a filesystem originally developed by SGI^{TM}. Development started in 1993, for the IRIX operating system. In 2000, it was ported to Linux, and made available on GNU/Linux distributions. The design goal of XFS is to achieve high scalability in terms of IO threads, number of disks, file/filesystem size. It is an overwrite class filesystem, which uses B-tree of extents to manage disk space. A journal is used to ensure metadata operation atomicity. Snapshots are not supported; an underlying volume manager is expected to support that operation.

ReiserFS [Reiser 2001] is a general purpose Linux filesystem, which inspired some of the BTRFS architecture and design. It was built by Hans Reiser and a team of engineers at Namesys^{TM}. It was the first journaled filesystem to be included in the standard Linux kernel, and it was the default filesystem on many Linux distributions for a number of years. ReiserFS uses a single tree to hold the entire filesystem, instead of separate trees for file and directory. In order to reduce internal fragmentation, *tail packing* is implemented. The main idea is to pack the tail, the last partial block, of multiple files into a single block.

LFS [Rosenblum and Ousterhout 1992] introduced the concept of log-structured storage in the early 1990s. The main idea is to write all modifications to disk in large chunks, called *segments*. The filesystem can be thought of as a large tree that has a well-known root, but whose data is spread across the live segments. When disk space runs low, a *segment cleaner* background process kicks in. It frees segments by merging several partly full segments, into a smaller number of full segments. In terms of crash recovery, the live segments can be thought of as a log that could be replayed, reproducing the filesystem state immediately prior to the crash. In order to reduce recovery time, the system takes periodic checkpoints. This limits replay to the last checkpoint plus every new segment since. Generally speaking, as long as overall disk utilization was below 80–90%, write performance was very good compared to contemporary filesystems, while read performance was comparable. The Achilles heal of LFS is a random write workload, followed by a sequential scan. Note that BTRFS, in contrast to LFS, can work well even in the absence of long free segments.

NILFS [Konishi et al.] is a modern implementation of an LFS inside the Linux kernel. It is developed by NTT^{TM} Japan, as an open-source project. The first version was released in 2005, and development has been ongoing since. NILFS brings efficient

B-tree indexes to the filesystem tree, efficient support for read-only snapshots, and fast recovery from crashes. Snapshot and crash recovery implementation hinge on the idea that the filesystem can be thought of as a long log, stretching from filesystem creation to present day. Data can be discarded only if it was overwritten, or if the snapshot it belonged to was discarded. Recovery is implemented by moving to the latest checkpoint, and then rolling forward. The main challenge for NILFS is writing a good garbage collector that takes into account snapshots. NILFS is a work in progress; features like FSCK, online defragmentation, writeable snapshots, and quotas remain to be implemented.

WAFL [Hitz et al. 1994] is the filesystem used in the NetApp^{TM} commercial file server; development started in the early 1990s. It is a copy-on-write filesystem that is especially suited for NFS [Callaghan et al. 1995; Shepler et al. 2000] and CIFS [Heizer et al. 1996] workloads. It uses NVRAM to store an operation log, and supports recovery up to the last acknowledged operation. This is important for supporting low-latency file write operations; NFS write semantics are that an operation is persistant once the server sends an acknowledgment to the client. WAFL manages space in 4KB blocks, and indexes files using a balanced tree structure. Snapshots are supported, as well as RAID. Free space is managed using a form of bitmaps. WAFL is a mature and feature rich filesystem.

2.1. ZFS

ZFS [Bonwick and Moore] is a copy-on-write filesystem originally developed by SUN^{TM} for its Solaris operating system. Development started in 2001, with the goal of replacing UFS, which was reaching its size limitations. ZFS was incorporated into Solaris in 2005. The ZFS creators; Jeff Bonwick and Matthew Ahrens, saw several problems in the filesystems that existed on Unix platforms at the time.

— Data would get corrupted on disk without the filesystem, without an underlying storage system to detect it. This would eventually cause bad data to be delivered to the application.
— Storage was split into many subcomponents that could not be managed as a single unit. For example, to construct a filesystem from many disks, one had to use a logical volume manager, build a virtual disk, and then lay a filesystem on top of it.
— There were many limitations on volume size, filesystem size, disk size, maximal disk size, maximal number of inodes, and so on.
— There was growing disparity between disk latency and bandwidth. This was not compensated for by new filesystem algorithms.

They decided to base their new system on the following engineering principles.

— *Pooled storage*. All the disks are managed as one large storage pool. There is no separate volume manager.
— *Tree of blocks*. The file-system logically looks like a large tree of blocks. Updates are performed with copy-on-write; they ripple up the tree. In order to commit a set of changes, all changes are written off to the side, and finally, the root is overwritten in an atomic step.
— *Checksums*. The tree has data at its leaves, and internal nodes containing pointers. The pointers contain checksums of the nodes they point to. This allows online verification, as one traverses down the tree.
— *Simple administration*. This can now be done, as volume management is integrated into the filesystem.

disk	D_1	D_2	D_3	D_4	P
row 1	d_{11}	d_{12}	d_{13}	d_{14}	p_{15}
row 2	d_{21}	d_{22}	d_{23}	d_{24}	p_{25}
row 3	d_{31}	d_{32}	d_{33}	d_{34}	p_{35}

Fig. 1. Three first stripes on a RAID4 configuration with 5 disks. Each row is a stripe, with the XOR of the data pages equal to the parity page.

IO size	data pages	parity page	configuration
4KB	d_{11}	p_{12}	1 + P
12KB	d_{13}, d_{14}, d_{15}	p_{21}	3 + P
8KB	d_{22}, d_{23}	p_{24}	2 + P
16KB	$d_{25}, d_{31}, d_{32}, d_{33}$	p_{34}	4 + P

Fig. 2. A series of IOs, and their RAID-Z stripes. Note that different configurations are created for different IO sizes.

disk	D_1	D_2	D_3	D_4	D_5
row 1	d_{11}	p_{12}	d_{13}	d_{14}	d_{15}
row 2	p_{21}	d_{22}	d_{23}	p_{24}	d_{25}
row 3	d_{31}	d_{32}	d_{33}	p_{34}	

Fig. 3. Layout of data and parity on disk, resulting from the IOs listed in Figure 2. Note that data and parity are spread more or less evenly across the disks.

ZFS uses a an interesting RAID algorithm, called *RAID-Z*. Traditional RAID-4 [Patterson et al. 1988] uses a configuration like $4 + P$. There are four data disks, D_1, through D_4, and a single parity disk P. Figure 1 shows an example for how pages are laid out. Three rows are presented; each row consists of four data pages and one parity page. The XOR of the data pages in a row is equal to the parity page: $d_{i1} \oplus d_{i2} \oplus d_{i3} \oplus d_{i4} = p_{i5}$.

Each row is also called a *full stripe*. As long as data is written to disk in full stripes, performance is good. However, updating a page in place requires a costly read-modify-write operation. For example, to update d_{11} we need to:

(1) read old values for d_{11} and p_{15};
(2) compute new parity $p'_{15} = d'_{11} \oplus d_{11} \oplus p_{15}$;
(3) write new values d'_{11} and p'_{15}.

An additional problem with RAID4 is the *write hole* problem. If a partial update is written to disk, and power is lost, then an entire row becomes garbage. This is resolved with NVRAM in hardware-based RAID controllers, and an additional log in software RAID.

In order to have updates that are always full stripe, RAID-Z uses an adaptive stripe size. Examine a situation with five disks. In order to provide fault tolerance to a single disk fault, RAID-Z will always add a parity page. This is done at different stripe widths, depending on IO size. For example, Figure 2 shows a series of writes, and Figure 3 shows the resulting disk layout.

RAID-Z has extensions to support double fault tolerance. In order to resolve the write-hole problem, ZFS does only copy-on-write updates, this allows committing a set of changes atomically without using expensive NVRAM or additional logging.

In order to support ZFS snapshots[1], block-pointers are enhanced with the child's *birth time*. This is not an actual wall clock, but rather the transaction ID used to write that block to disk. Having the file-system tree decorated with birth times allows efficient snapshot creation and deletion. It also allows creating writeable snapshots (clones). An important limitation however, is that a a clone is not a first class citizen—it cannot be cloned again.

While ZFS and BTRFS have a lot in common, they also differ substantially.

— *Goals*. ZFS is intended for an enterprise class Solaris server, BTRFS is intended to run on (almost) all things running Linux.
— *Filesystem check fsck*. ZFS does not have an fsck utility. If the RAID and checksum algorithms are insufficient, the ZFS user is expected to restore from external backup. By contrast, Linux is installed on many desktops and smartphones, that do not have enterprise grade backups. Linux users have therefore come to expect FSCK utilities. Unfortunately, writing a good recovery tool for BTRFS is very hard, as the data can move around on disk, and there is no fixed location for metadata (except for the superblock).
— *Blocks vs. extents*. ZFS uses blocks that are powers of two: 4KB, 8KB, ..., 128KB. Each object has a fixed block size. the RAID-Z code works well with fixed sized blocks, however since there are several different block sizes, we could run out of a particular size. This requires a procedure called *ganging*, where larger blocks are created from smaller blocks. Since there might not be contiguous subblocks, the system can very well get into trouble. On BTRFS, extents are used instead of blocks. This eliminates the need for special block sizes. However, if the system is left with a highly fragmented free space, writes to disk are still going to be inefficient.
— *Checksums*. Both ZFS and BTRFS calculate checksums. However, ZFS keeps them in the block-pointers, whereas BTRFS keeps data checksums in a separate tree. This is necessary for BTRFS, since extents could be very large, and we would like to be able to validate each page separately.
— *Snapshots vs. clones*. ZFS uses birth-times to build snapshots. BTRFS uses the reference-counting mechanism instead. The result is very similar, however, the BTRFS mechanism is more general, in that it supports clones as first class citizens.
— *RAID*. ZFS uses RAID-Z, and supports several RAID levels including single/dual parity (RAID $\{5, 6\}$). BTRFS uses something closer to a standard RAID layout; support is currently available only for mirroring, striping, and mirroring+striping (RAID levels $\{0, 1, 10\}$).
— *Back references*. ZFS does not support back-references, while BTRFS does. ZFS is assumed to run in a big server with many disks, and it can be assumed that redundancy through RAID-Z is implemented. This means that ZFS can recover bad blocks through reconstruction; failing that, there is backup. BTRFS cannot assume underlying RAID and multiple devices, hence it must be able to cope with bad blocks by other means.
— *Deduplication*. ZFS supports deduplication, although this comes at a very significant memory cost. BTRFS does not support this feature at the moment. Due to the memory requirements, it might be a feature only fit for high end servers.

2.2. Log Structured Merge Trees

Log Structured Merge Trees (LSM-trees) [O'Neil et al. 1996] are an on-disk data structure designed to support high update rates. An in-memory table maintains the

[1]https://blogs.oracle./com/ahrens/is_it_magic

latest set of updates. When it fills, it is sorted and flushed to disk as a subtable. Periodically, the on-disk tables are compacted. This kind of data structure is now used by Google^{TM} in its big-table implementation [Chang et al. 2006; Dean and Ghemawat], Yahoo^{TM} [Sears and Ramakrishnan 2012] in its PNUTS platform, and others. It is useful where the update rate is high. An additional constraint is that read accesses should mostly be point accesses, and not range queries. This is because point accesses to the wrong subtable can easily be discarded using Bloom filters.

At this point, it is hard to see how LSMs could serve as the main filesystem data structure. A general purpose filesystem must provide excellent support for random read workloads. In addition, reading from file maps to a query for allocated pages in a range; not to a point query. One possibility, is auxiliary use, for example Macko et al. [2010] uses LSM trees to track back references in a filesystem. Another possibility is presented in Ren and Gibson [2012], where only small files are stored in the LSM.

Recently published work [Shetty et al. 2013], shows how to combine the benefits of B-trees, LSM trees, and avoid most of the pitfalls.

It remains to be seen what role LSM trees will play in filesystem layouts. It is a topic that is currently attracting a lot of attention.

3. FUNDAMENTALS

Filesystems support a wide range of operations and functionality. A full description of all the BTRFS options, use cases, and semantics would be prohibitively long. The focus of this work is to explain the core concepts, and we limit ourselves to the more basic filesystem operations: file create/delete/read/write, directory lookup/iteration, snapshots and clones. We also discuss data integrity, crash recovery, and RAID. Our description reflects the filesystem at the time of writing.

The following terminology is used throughout.

— *Page, block*: a 4KB contiguous region on disk and in memory. This is the common page size for Intel^{TM} architectures. On other architectures, for example PowerPC (PPC), the page size could be 8KB, or even larger. To simplify the discussion, we assume the page size here is 4KB.
— *Extent*: a contiguous on-disk area. It is page-aligned, and its length is a multiple of pages.
— *Copy-on-write (COW)*: creating a new version of an extent or a page at a different location. Normally, the data is loaded from disk to memory, modified, and then written elsewhere. The idea is not to update the original location in place, risking a power failure and partial update.

3.1. COW Friendly B-Trees

COW friendly B-trees are central to the BTRFS data-structure approach. For completeness, this section provides a recap of how they work. For a full account, the interested reader is referred to: Rodeh [2006a, 2006b, 2008, 2010]. The main idea is to use standard B+-tree construction [Comer 1979] but, (1) employ a top-down update procedure, (2) remove leaf-chaining, and (3) use lazy reference-counting for space management.

For purposes of this discussion, we use trees with short integer keys, and no actual data items. The B-tree invariant is that a node can maintain 2 to 5 elements before being split or merged. Tree nodes are assumed to take up exactly one page. Unmodified pages are rectangle-shaped with an orange border, and COWed pages are hexagon shaped with a green border.

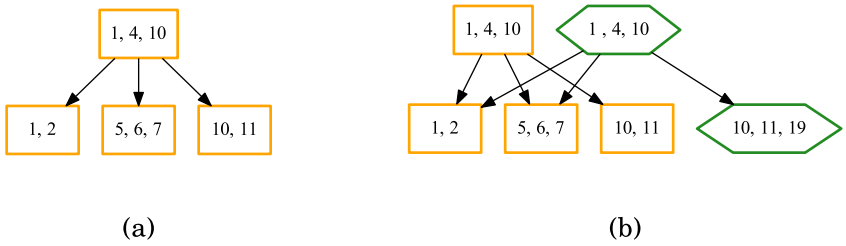


Fig. 4. (a) A basic B-tree (b) Inserting key 19, and creating a path of modified pages.

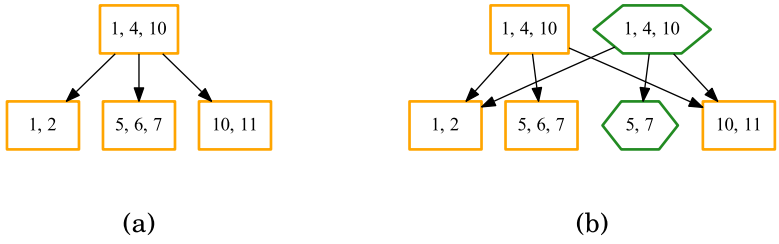


Fig. 5. (a) A basic tree, (b) Deleting key 6.

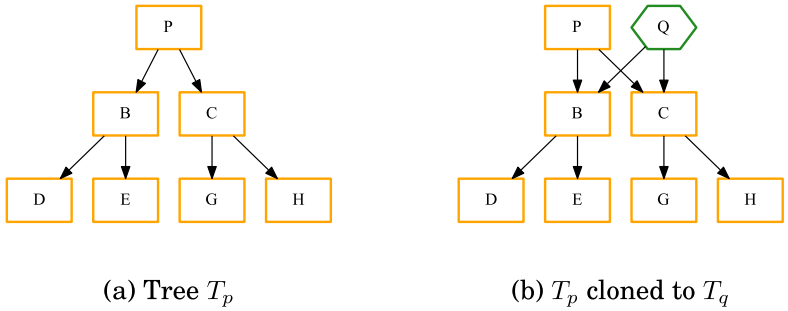


Fig. 6. Cloning tree T_p. A new root Q is created, initially pointing to the same nodes as the original root P. As modifications are applied, the trees will diverge.

Figure 4(a) shows an initial tree with two levels. Figure 4(b) shows an insert of a new key, 19 into the right-most leaf. A path is traversed down the tree, and all modified pages are written to new locations, without modifying the old pages.

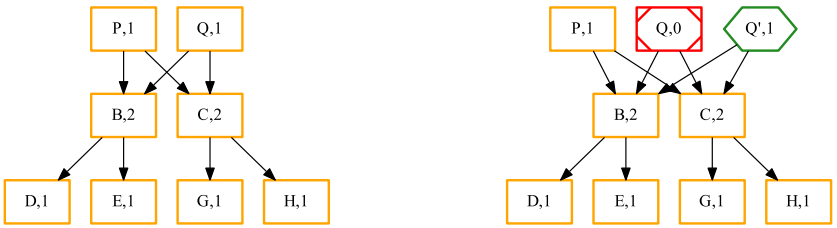
In order to remove a key, copy-on-write is used. Remove operations do not modify pages in place. For example, Figure 5 shows how key 6 is removed from a tree. Modifications are written off to the side, creating a new version of the tree.

In order to clone a tree, its root node is copied, and all the child pointers are duplicated. For example, Figure 6 shows a tree T_p, that is cloned to tree T_q. Tree nodes are denoted by symbols. As modifications are applied to T_q, sharing will be lost between the trees, and each tree will have its own view of the data. Cloning a node, is very similar to COW, except that the original node is not marked for erasure.

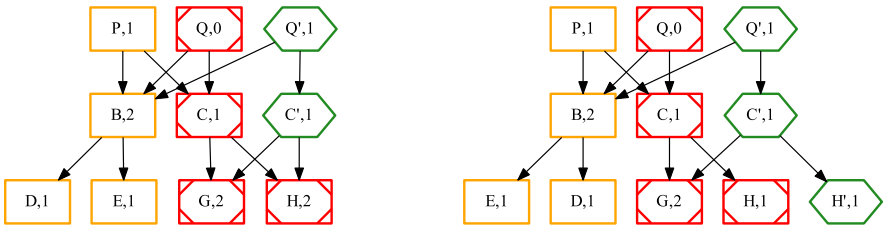
Since tree nodes are reachable from multiple roots, garbage collection is needed for space reclamation. In practice, file systems are directed acyclic graphs (DAGs). There are multiple trees with shared nodes, but there are no cycles. Therefore, reference-counters (*ref-counts*) can be, and are, used to track how many pointers there are to tree nodes. A ref-count for a node records how many direct pointers there are to it. Once the counter reaches zero, a block can be reused. The ref-counts are not stored in the nodes,

(a) Tree T_p (b) T_p cloned to T_q

Fig. 7. Cloning tree T_p. A new root Q is created, initially pointing to the same nodes as the original root P. The ref-counts for the immediate children are incremented. The grandchildren remain untouched.



(a) Initial trees, T_p and T_q (b) Shadow Q

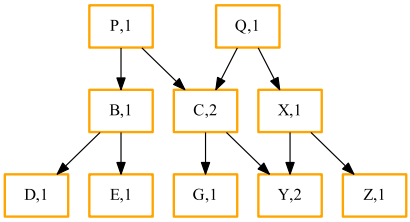


(c) shadow C (d) shadow H

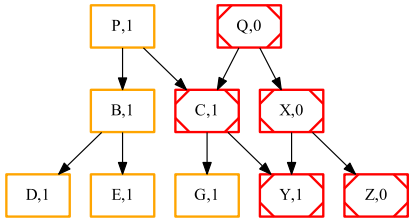
Fig. 8. Inserting a key into node H of tree T_q. The path from Q to H includes nodes $\{Q, C, H\}$, these are all COWed. Sharing is broken for nodes C and H; the ref-count for C is decremented.

rather the system keeps them stored persistently in a separate data structure (see the extent-tree in Sections 3.4, 3.5). This allows modifying a node's counter, without having to modify the node itself. In order to keep track of ref-counts, the copy-on-write mechanism is modified. Whenever a node is COWed, the ref-count for the original is decremented, and the ref-counts for the children are incremented. Figure 7 shows an example of a clone operation, with a ref-count indication. The convention is that diamond-like nodes with red borders are unchanged except for their ref-count. Tree T_P is cloned, causing a new copy of the root to be created, and an increment in the ref-counts of the immediate children $\{B, C\}$.

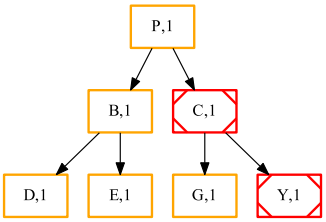
Figure 8 shows an example of an insert-key into leaf H, tree q. The nodes on the path from Q to H are $\{Q, C, H\}$. They are all modified and COWed.



(a) Initial trees T_p and T_q



(b) Deleting T_q



(c) Removing unallocated nodes

Fig. 9. Deleting a tree rooted at node Q. Nodes $\{X, Z\}$, reachable solely from Q, are deallocated. Nodes $\{C, Y\}$, reachable also through P, have their ref-count reduced from 2 to 1.

Figure 9 shows an example of a tree delete. The algorithm used is a recursive tree traversal, starting at the root. For each node N, we have the following.

— ref-count(N) $>$ 1. Decrement the ref-count and stop downward traversal. The node is shared with other trees.
— ref-count(N) $==$ 1. It belongs only to q. Continue downward traversal and deallocate N.

3.2. Checkpoints

The filesystem comprises a forest of trees that are all modified using copy-on-write. Updates are accumulated in memory, and written out in atomic *checkpoints*. This process is fully explained in Section 3.4. Checkpoints are labeled with a monotonically increasing *generation number*, which is embedded in various on-disk data structures. This is a logical timer that can be used for validation, or difference calculations. Note that some operations, such as `fsync` have special handling, so as to avoid full checkpoints; see Section 3.6.

3.3. Filesystem B-Tree

The BTRFS B-tree is a generic data structure that knows only about three types of data structures: keys, items, and block headers. The block header is fixed size and holds fields like checksums, flags, filesystem IDS, generation number, and so on. A key describes an object address using the following structure.

```
struct btrfs_key {
    u64: objectid;  u8: type;  u64 offset;
}
```

Fig. 10. A leaf node with three items. The items are fixed size, but the data elements are variably sized.

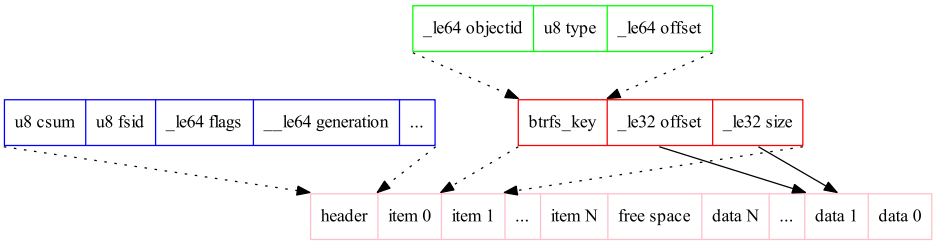


Fig. 11. A detailed look at a generic leaf node holding keys and items.

An item is a BTRFS key with additional offset and size fields:

```
struct btrfs_item {
    struct btrfs_key key; u32 offset; u32 size;
}
```

Internal tree nodes hold only `[key, block-pointer]` pairs. Leaf nodes hold arrays of `[item, data]` pairs. Item data is variably sized. A leaf stores an array of items in the beginning, and a reverse sorted data array at the end. These arrays grow towards each other. For example Figure 10 shows a leaf with three items $\{I_0, I_1, I_2\}$ and three corresponding data elements $\{D_2, D_1, D_0\}$.

Item data is variably sized, and various filesystem data structures are defined as different types of item data. The type field in the key indicates the type of data stored in the item.

The filesystem is composed of objects, each of which has an abstract 64bit *object_id*. When an object is created, a previously unused object_id is chosen for it. The object_id makes up the most significant bits of the key, allowing all of the items for a given filesystem object to be logically grouped together in the B-tree. The type field describes the kind of data held by an item; an object typically comprises several items. The offset field describes data held in an extent.

Figure 11 shows a more detailed schematic of a leaf node.

Inodes are stored in an inode item at offset zero in the key, and have a type value of one. Inode items are always the lowest valued key for a given object, and they store the traditional stat data for files and directories. This includes properties such as size, permissions, flags, and a count of the number of links to the object. The inode structure is relatively small, and will not contain embedded file data, extended attribute data, or extent mappings. These things are stored in other item types.

Small files that occupy less than one leaf block may be packed into the B-tree inside the extent item. In this case the key offset is the byte offset of the data in the file, and the size field of the item indicates how much data is stored. There may be more than one of these per file.

Larger files are stored in extents. These are contiguous on-disk areas that hold user data without additional headers or formatting. An extent-item records a generation number recording when the extent was created, a `[disk block, length]` pair to record the area on disk, and a `[logical file offset, length]` pair to record the file area. An extent is a mapping from a logical area in a file, to a physical area on disk. If a file is stored in a small number of large extents, then a common operation such as a full file read will be efficiently mapped to a few disk operations. This explains the attractiveness of extents.

Some filesystems use fixed size blocks instead of extents. Using an extent representation is much more space efficient, however this comes at the cost of more complexity. For example, examine a file with data in the range `0–64KB`, where the file is laid out optimally as one contiguous disk area. With a 4KB block representation, we will need 16 pointers to represent it. If we overwrite the area `8–16KB`, we will still need 16 pointers to represent the file. With an extent representation, the file will initially consist of exactly one 64KB extent. The overwrite operation however, muddies the water. The 8–16KB area must be written elsewhere, due to copy-on-write considerations. This splits the file into three extents: `0–8KB`, `8–16KB`, `16–64KB`.

A directory holds elements of type `dir_item`. A dir_item is a structure comprising a file-name (string) and a BTRFS_key specifying the location of the object. The directory contains two sorted lists of dir_items. The first list holds all mappings, sorted by file name hash; it is used to satisfy path lookups. The second list is a copy of the first, but sorted by inode sequence number. It is used for bulk directory operations. The inode sequence number is stored in the directory, and is incremented every time a new file or directory is added. It approximates the on-disk order of the underlying file inodes, and thus saves disk seeks when accessing them. Bulk performance is important for operations like backups, copies, and filesystem validation. For example, `fsck` uses the iteration index to gain speed improvements as it needs to scan through the entire metadata with as few seeks as possible.

Figure 12 shows an example directory structure containing a root and three files: `player.c`, `player`, and `doc.txt`. The files have corresponding inode numbers: 259, 260, and 261. The root directory holds the files sorted by hashes; these are the DIR_ITEMs, their offset field holds the filename hash. The file names are also sorted by inode creation time, these are the INDEX_ITEMs, where the offset field contains the inode sequence number. The files themselves consist of an inode element, of which we show only the size attribute, and extents pointing to data. The `doc.txt` file is small enough to be allocated inside the metadata node, while the other files have extents allocated externally. For example, in order to sequentially read file `player.c`, the top level directory is searched for the hash of `player.c`. This is achieved by a B-tree search for a DIR_ITEM with `inode=256` and `offset=5012`. The resulting inode is 259, and then the extents for inode 259 can be read sequentially.

3.4. A Forest

A filesystem is constructed from a forest of trees. A superblock located at a fixed disk location is the anchor. It points to a *tree of tree roots*, which indexes the B-trees making up the filesystem. The trees are the following.

— *Subvolumes* store user visible files and directories. Each subvolume is implemented by a separate tree. Subvolumes can be snapshotted and cloned, creating additional B-trees. The roots of all subvolumes are indexed by the tree of tree roots.
— *Extent allocation tree* tracks allocated extents in `extent items`, and serves as an on-disk free-space map. All *back-references* to an extent are recorded in the extent item. This allows moving an extent if needed, or recovering from a damaged disk block. Taken as a whole, back-references multiply the number of filesystem disk pointers by two. For each forward pointer, there is exactly one back-pointer. See more details on this tree below.
— *Checksum tree* holds one `checksum item` per allocated extent. The item contains a list of checksums per page in the extent.
— *Chunk and device trees* are the indirection layer for handling physical devices. Allows mirroring/striping and RAID. Section 4 shows how multiple device support is implemented using these trees.

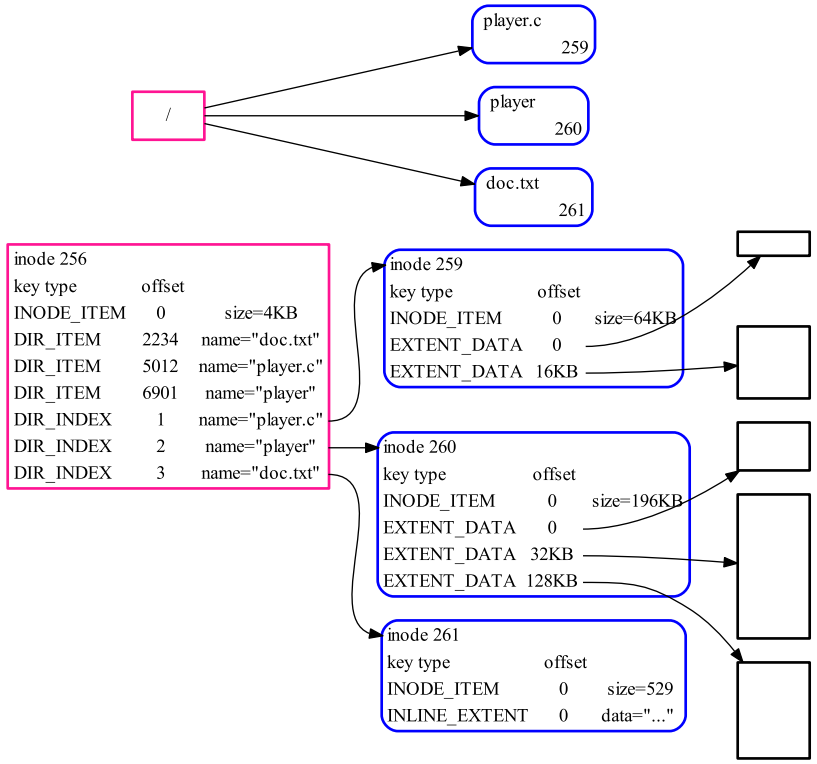


Fig. 12. A directory structure with three files: `player.c`, `player`, and `doc.txt`. The files have corresponding inode numbers: 259, 260, and 261. The filesystem tree holds a directory with a double mapping, and three file elements. The `doc.txt` file is small enough to be inlined, while the other files have external extents.

—*Reloc tree* is for special operations involving moving extents. Section 5 describes how the reloc-tree is used for defragmentation.

For example, Figure 13(a) shows a high-level view of the structure of a particular filesystem. The reloc and chunk trees are omitted for simplicity. Figure 13(b) shows the changes that occur after the user wrote to the filesystem.

Modifying user-visible files and directories causes page and extent updates. These ripple up the subvolume tree to its root. Changes also occur to extent allocation, ref-counts, and back-pointers. These ripple through the extent tree. Data and metadata checksums change, these updates modify the checksum tree leaves, causing modifications to ripple up. All these tree modifications are captured at the topmost level as a new root in the tree of tree roots. Modifications are accumulated in memory, and after a timeout, or enough pages have changed, are written in batch to new disk locations, forming a *checkpoint*. The default timeout is 30 seconds. Once the checkpoint has been written, the superblock is modified to point to the new checkpoint; this is the only case where a disk block is modified in place. If a crash occurs, the filesystem recovers by reading the superblock, and following the pointers to the last valid on-disk checkpoint. When a checkpoint is initiated, all dirty memory pages that are part of it are marked immutable. User updates received while the checkpoint is in flight cause immutable pages to be re-COWed. This allows user visible filesystem operations to proceed without damaging checkpoint integrity. Checkpoints are numbered by a monotonically increasing *generation number*, which is embedded in various on disk data structures.

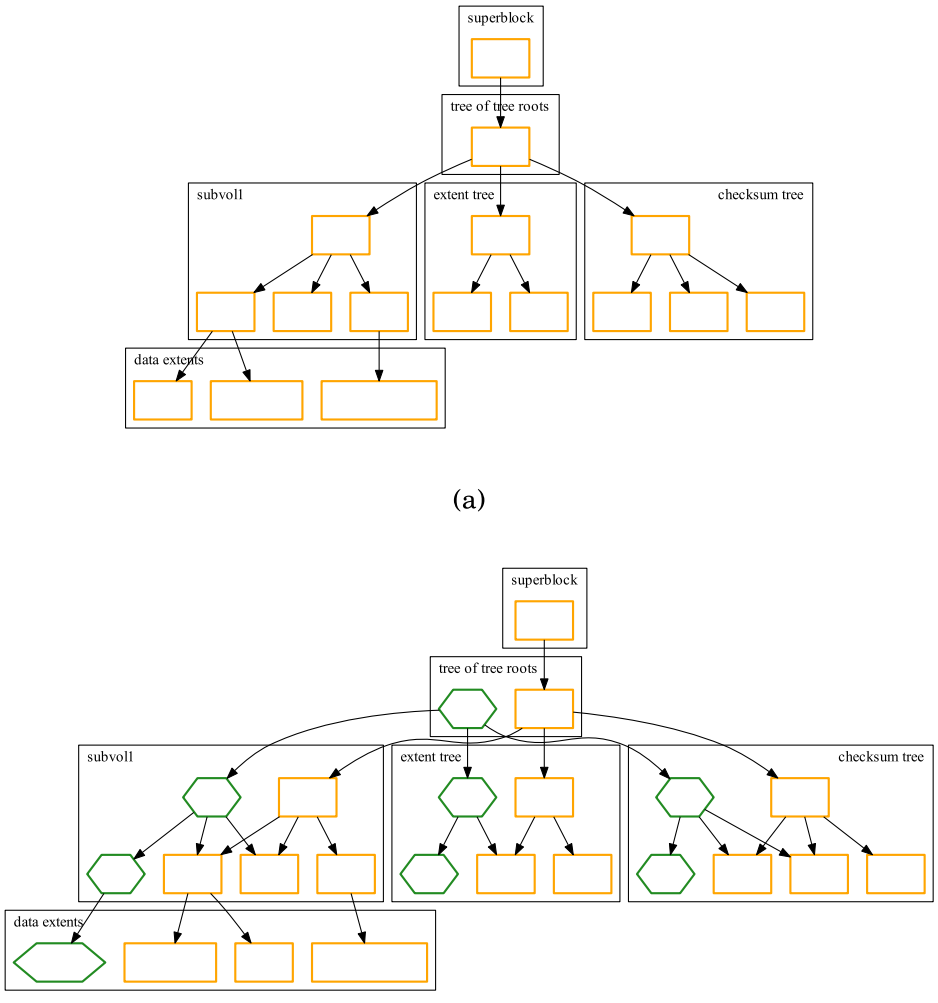


Fig. 13. (a) A filesystem forest. (b) The changes that occur after modification; modified pages are colored green.

Subvolume trees can be snapshoted and cloned, and they are therefore ref-counted. All other trees keep metadata per disk range, and they are never snapshoted. Reference counting is unnecessary for them.

A filesystem update affects many on-disk structures. For example, a 4KB write into a file changes the file i-node, the file-extents, checksums, and back-references. Each of these changes causes an entire path to change in its respective tree. If users performed entirely random updates, this would be very expensive for the filesystem. Fortunately, user behavior normally has a lot of locality. If a file is updated, it would be updated with lots of new data; files in the same directory have a high likelihood of coaccess. This allows coalescing modified paths in the trees. Nonetheless, worst cases are considered in the filesystem code. Tree structure is organized so that file operations normally

File	On disk extents
foo	100-128KB
bar	100-110KB, 300-310KB, 120-128KB

Fig. 14. Foo and its clone bar share parts of the extent 100–128KB. Bar has an extent in the middle that has been overwritten and is now located much further away on disk

modify single paths. Large-scale operations are broken into parts, so that checkpoints never grow too large. Finally, special block reservations are used so that a checkpoint will always have a home on disk, guaranteeing forward progress.

Using copy-on-write as the sole update strategy has pros and cons. The upside is that it is simple to guarantee operation atomicity, and data-structure integrity. The downside is that performance relies on the ability to maintain large extents of free contiguous disk areas. In addition, random updates to a file tend to fragment it, destroying sequentiality. A good defragmentation algorithm is required; this is described in Section 5.

Checksums are calculated at the point where a block is written to disk. At the end of a checkpoint, all the checksums match, and the checksum at the root block reflects the entire tree. Metadata nodes record the generation number when they were created. This is the serial number of their checkpoint. B-tree pointers store the expected target generation number, which allows detection of phantom or misplaced writes on the media. Generation numbers and checksums serve together to verify on-disk block content.

3.5. Extent Allocation Tree

The extent allocation tree holds `extent-items`, each describing a particular contiguous on-disk area. There could be many references to an extent, each addressing only part of it. For example, consider file `foo`, which has an on-disk extent `100KB–128KB`. File `foo` is cloned creating file `bar`. Later on, a range of 10KB is overwritten in `bar`. This could cause the following situation.

There are three pointers into extent 100–128KB, covering different parts of it. The extent-item keeps track of all such references, to allow moving the entire extent at a later time. An extent could potentially have a large number of back references, in which case the extent-item does not fit in a single B-tree leaf node. In such cases, the item spills and takes up more than one leaf.

A back reference is a logical hint—it is not a physical address. It is constructed from the root_object_id, generation_id, tree level, and lowest object-id in the pointing block. This allows finding the pointer after a lookup traversal starting at the root object-id. In this example, extent `100–128KB` has three back references, one from `foo`, and two from `bar`. The back references also serve as the ref-count; when it reaches zero, the extent can be freed.

An important operation on the extent-tree is finding all references into a disk area. The extent items are sorted by start address, and there are no overlaps between items. This allows doing a range query on the B-tree, resulting in all extents in the range. We can then extract all the back references and find all pointers to this disk area. This is a crucial operation used for moving data out of an area. There could be multiple reasons for this: garbage collection, device removal, file system resize, RAID rebalance, and so on.

3.6. Fsync

`fsync` is a operation that flushes all dirty data for a particular file to disk. An important use case is by databases that wish to ensure that the database log is on disk, prior to

committing a transaction. Latency is important; a transaction will not commit until the log is fully on disk. A naive fsync implementation is to checkpoint the entire filesystem. However, that suffers from high latency. Instead, modified data and metadata related to the particular file are written to a special *log-tree*. Should the system crash, the log-tree will be read as part of the recovery sequence. This ensures that only minimal and relevant modifications will be part of the fsync code path.

3.7. Compression

Compression is implemented at the extent level. Two compression algorithms are supported: ZLIB [Gailly and Adler], and LZO[2]. ZLIB is slower, but provides higher compression ratios. LZO is faster, but provides a worse compression ratio.

3.8. Concurrency

Modern systems have multiple CPUs with many cores. Taking advantage of this computing power through parallelism is an important consideration.

Old generations are immutable on disk and their access does not require locking. The in-memory under-modification pages require protection. Since data is organized in trees, the strategy is to use a read/write locking scheme. Tree traversal is initiated in read mode. When a node that requires update is encountered, the lock is converted to write mode. If a block B requires COW, traversal is restarted. The new traversal stops at $parent(B)$, COWs B, modifies the parent pointer, and continues down.

Tree traversals are top-down. They start at the top, and walk down the tree, it is unnecessary to walk back up.

4. MULTIPLE DEVICE SUPPORT

Linux has two device management subsystems: the *device mapper* (DM) and the *software RAID subsystem* (MD). The device mapper is a stackable design starting with raw devices, and building up with additional modules supporting multipathing, thin provisioning, mirroring, striping, RAID5/6, and so on. The software RAID module supports various fault tolerance levels by combining lower level raw devices. Both modules' primary function is to take raw disks, merge them into a virtually contiguous block-address space, and export that abstraction to higher-level kernel layers. An important issue is that checksums are not supported, which causes a problem for BTRFS. For example, consider a case where data is stored in RAID-1 form on disk and each 4KB block has an additional copy. If the filesystem detects a checksum error on one copy, it needs to recover from the other copy. DMs hide that information behind the virtual address space abstraction and return one of the copies. To circumvent this problem, BTRFS does its own device management. It calculates checksums, stores them in a separate tree, and is then better positioned to recover data when media errors occur.

A machine may be attached to multiple storage devices; BTRFS splits each device into large *chunks*. The rule of thumb is that a chunk should never be more than 10% of the device size. At the time of writing 1GB chunks are used for data, and 256MB chunks are used for metadata.

A *chunk tree* maintains a mapping from logical chunks to physical chunks. A *device tree* maintains the reverse mapping. The rest of the filesystem sees logical chunks, and all extent references address logical chunks. This allows moving physical chunks

[2]http://en.wikipedia.org/wiki/Lempel-Ziv-Oberhumer

logical chunks	disk 1	disk 2	disk 3
L_1	C_{11}	C_{21}	
L_2		C_{22}	C_{31}
L_3	C_{12}		C_{32}

Fig. 15. To support RAID1 logical chunks, physical chunks are divided into pairs. Here there are three disks, each with two physical chunks, providing three logical chunks. Logical chunk L_1 is built out of physical chunks C_{11} and C_{21}.

logical chunks	disk 1	disk 2	disk 3
L_1	C_{11}	C_{21}	
L_2		C_{22}	C_{31}
L_3	C_{12}	C_{23}	
L_4		C_{24}	C_{32}

Fig. 16. One large disk and two small disks in a RAID1 configuration.

logical chunks	disk 1	disk 2	disk 3	disk 4
L_1	C_{11}	C_{21}	C_{31}	C_{41}
L_2	C_{12}	C_{22}	C_{32}	C_{42}
L_3	C_{13}	C_{23}	C_{33}	C_{43}

Fig. 17. Striping with four disks, stripe width is $n = 4$. Three logical chunks are each made up of four physical chunks.

under the covers without the need to backtrace and fix references. The chunk/device trees are small, and can typically be cached in memory. This reduces the performance cost of an added indirection layer.

Physical chunks are divided into groups according to the required RAID level of the logical chunk. For mirroring, chunks are divided into pairs. Figure 15 presents an example with three disks, and groups of two. For example, logical chunk L_1 is made up of physical chunks C_{11} and C_{21}. Figure 16 shows a case where one disk is larger than the other two.

For striping, groups of n chunks are used, where each physical chunk is on a different disk. For example, Figure 17 shows a stripe width of four ($n = 4$), with four disks and three logical chunks.

At the time of writing, RAID levels 0,1, and 10 are supported. Support for RAID5/6 has recently been added (BTRFS version 3.9). The core idea in higher RAID levels is to use chunk groups with Reed-Solomon [Reed and Solomon 1960] parity relationships. For example, Figure 18 shows a RAID6 configuration where logical chunks $L_{1,2,3}$ are constructed from doubly protected physical chunks. For example, L_1 is constructed from $\{C_{11}, C_{21}, C_{31}, C_{41}\}$. Chunks $\{C_{11}, C_{12}\}$ hold data in the clear, $C_{31} = C_{21} \oplus C_{11}$, and $C_{41} = Q(C_{21}, C_{11})$. Function Q is a defined by Reed-Solomon codes such that any double chunk failure combination would be recoverable.

Replicating data and storing parity is costly overhead for a storage system. However, it allows recovery from many media error scenarios. The simplest case is RAID1, where each block has a mirror copy. When the filesystem tries to read one copy and discovers an IO or checksum error, it tries the second copy. If the second copy also has an error, then the data is lost. Back references have to be traced up the filesystem tree, and the file has to be marked as damaged. If the second copy is valid, then it can be returned to the caller. In addition, the first copy can be overwritten with the valid data. A proactive approach, where a low intensity scrub operation is continuously run on the data, is also supported.

	physical	disks		
logical chunks	D_1	D_2	P	Q
L_1	C_{11}	C_{21}	C_{31}	C_{41}
L_2	C_{12}	C_{22}	C_{32}	C_{42}
L_3	C_{13}	C_{23}	C_{33}	C_{43}

Fig. 18. A RAID6 example. There are four disks, $\{D_1, D_2, P, Q\}$. Each logical chunk has a physical chunk on each disk. For example, the raw data for L_1 is striped on disks D_1 and D_2. C_{31} is the parity of C_{11} and C_{21}, C_{41} is the calculated Q of chunks C_{11} and C_{12}.

(a) 2 disks	logical chunks	disk 1	disk 2	
	L_1	C_{11}	C_{21}	
	L_2	C_{12}	C_{22}	
	L_3	C_{13}	C_{23}	
(b) disk added				disk 3
	L_1	C_{11}	C_{21}	
	L_2	C_{12}	C_{22}	
	L_3	C_{13}	C_{23}	
(c) rebalance				
	L_1	C_{11}	C_{21}	
	L_2		C_{22}	C_{12}
	L_3	C_{13}		C_{23}

Fig. 19. Device addition. Initially, (a) there are two disks. In state (b), another disk is added, which is initially empty. State (c) shows the goal: data spread evenly on all disks. Here, physical chunks C_{12} and C_{23} were moved to disk 3.

There is flexibility in the RAID configuration of logical chunks. A single BTRFS storage pool can have various logical chunks at different RAID levels. This decouples the top-level logical structure from the low-level reliability and striping mechanisms. This is useful for operations such as:

(1) changing RAID levels on the fly, increasing or decreasing reliability;
(2) changing stripe width: more width allows better bandwidth;
(3) giving different subvolumes different RAID levels. Perhaps some subvolumes require higher reliability, while others need more performance at the cost of less reliability.

The default behavior is to use RAID1 for filesystem metadata, even if there is only one disk. This gives the filesystem a better chance to recover when there are media failures.

Common operations that occur in the lifetime of a filesystem are device addition and removal. This is supported by a general balancing algorithm, which tries to spread allocations evenly on all available disks, even as the device population changes. This is carried out as a background process, with minimal interference with user IO. For example, in Figure 19(a) the system has two disks in a RAID1 configuration; each disk holds $\frac{1}{2}$ of the raw data. Then, a new disk is added (see Figure 19(b)). The goal of the balancing code is to reach the state shown in Figure 19(c), where data is spread evenly on all three disks, and each disk holds $\frac{1}{3}$ of the raw data.

When a device is removed, the situation is reversed. From a $\frac{1}{3}$ ratio (as in Figure 19(c)), the system has to move back to $\frac{1}{2}$ ratio. If there are unused chunks on the remaining disks, then the rebalancer can complete the task autonomously. However, we are not always that fortunate. If data is spread across all chunks, then trying

to evict a chunk requires traversal through the filesystem, moving extents, and fixing references. This is similar to defragmentation, which is described in Section 5.

5. DEFRAGMENTATION AND RELOCATION

Fragmentation occurs when a filesystem runs low on long contiguous disk extents. This is a problem for all filesystems, which is especially acute for copy-on-write systems, because they write all new data to fresh disk areas. This means that even overwrites, IOs that update areas that have already been allocated, will require new allocation.

In BTRFS, updates are accumulated in memory, and written to disk at sync time. This allows batching optimizations, such as delayed allocation, and colocating data from the same file. Since the sync timer is normally about 30 seconds, and memory buffer space is limited, the batching process is helpful, but is not sufficient in itself to prevent fragmentation. There are several different approaches and algorithms for dealing with disk fragmentation. Note that, unlike traditional LFS [Rosenblum and Ousterhout 1992], BTRFS can continue to function in the absence of long disk extents. It can allocate data and metadata on many short extents if it has to. However, performance can get very bad in these kinds of situations.

The *nocow* option can be set on an entire filesystem or a particular file. It cancels copy-on-write for data blocks, unless there is a snapshot. COW still applies to metadata blocks. NOCOW makes sense for workloads where COW would be very expensive, for example, database workloads that do random small updates, followed by sequential scans. The normal BTRFS policy would write the blocks to disk in update order, which would cause very bad performance in the sequential scan. Databases could well be large enough to overwhelm the in-memory buffers, defeating the attempt to lay out the data in increasing address order.

For applications less demanding than databases, the user can set the *autodefrag* mount option for the filesystem. Under the autodefrag policy, the system continuously looks for files that are good candidates and schedules them for defragmentation. In order to defrag a file, it is read, COWed, and written to disk in the next checkpoint. This is likely to make it much more sequential, because the allocator will try to write it out in as few extents as possible. The downside is that sharing with older snapshots is lost. While this option is simple, and works well for many systems, it would be better to maintain sharing. This is currently a topic of active development. Autodefrag is intended to be a default mount option.

The policies used at the time of writing by autodefrag are as follows. If there is a small write (less than 64k) into the middle of the file the file is added to an autodefrag list and then it is defragged whenever the cleaner thread runs, which is awakened at the end of every transaction commit, every 30 seconds. This works well for workloads like Sqllite databases. Chrome or Firefox users will notice a speedup in startup time if they use autodefrag, since both applications use Sqllite.

The last important case is shrinking a filesystem, or evicting data from a disk. In this case, a *relocator* is used. This is an algorithm that scans a chunk and moves the live data off of it, while maintaining sharing. Relocation is a complex procedure, however, and disregarding sharing could cause an increase in space usage, at the point where we were trying to reduce space consumption.

The relocator works on a chunk by chunk basis. The general scheme is as follows.

(1) Move out all live extents (in the chunk).
(2) Find all references into a chunk.
(3) Fix the references while maintaining sharing.

The copy-on-write methodology is used throughout; references are never updated in place. Figure 20 shows a simple example. Extent K, marked as an oval with a blue

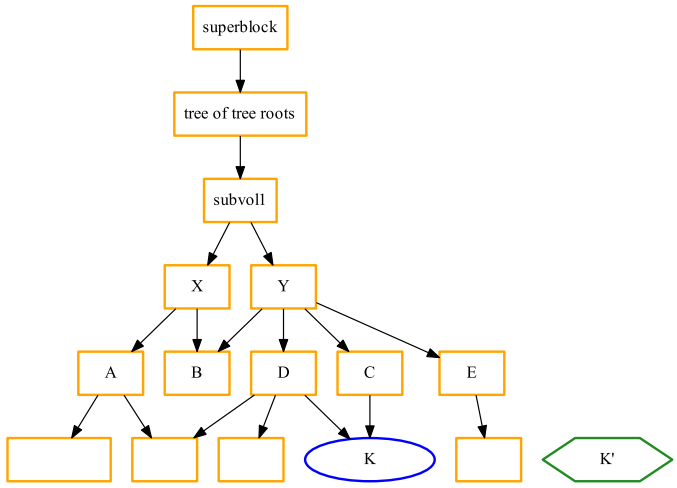


Fig. 20. Do a range lookup in the extent tree, find all the extents located in the chunk, and copy all the live extents to new locations.

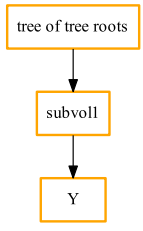


Fig. 21. Upper-level nodes stored in a backref_cache.

border, needs to be moved out of the chunk. It is copied out to K', which is on a different chunk.

In order to speed up some of the reference tracking, we follow back-references to find all upper-level tree blocks that directly or indirectly reference the chunk. The result is stored in a DAG-like data structure called a *backref_cache* (see Figure 21).

A list of subvolume trees that reference the chunk from the backref_cache is calculated; these trees are subsequently cloned (see Figure 22). This operation has two effects: (1) it freezes in time the state of the subvolumes, (2) it allows making off-to-the-side modifications to the subvolumes, while affording the user continued operation. The active filesystem continues to update the subvolumes; conflicting updates will be merged later on in the process.

Next, all the references leading to the chunks are followed, using back-references. COW is used to update them in the reloc trees (see Figure 23).

The last step is to merge the reloc trees with the originals. We traverse the trees and we find subtrees that are modified in the reloc tree, but where corresponding parts in the fs tree are not modified. These subtrees in the reloc tree, are swapped with their older counterparts from the fs tree. The end result is new fs-trees. Conflicts could arise if intermediate node N was updated by the active filesystem, while a new version (N') was created by the relocation code. We treat this by prefering the version of N from the active filesystem, and look for shared subtrees in the children of N. The worst case is where all intermediate nodes have been concurrently modified, leaving the relocated data extents orphan. In this case, the merge step will COW the immediate parents

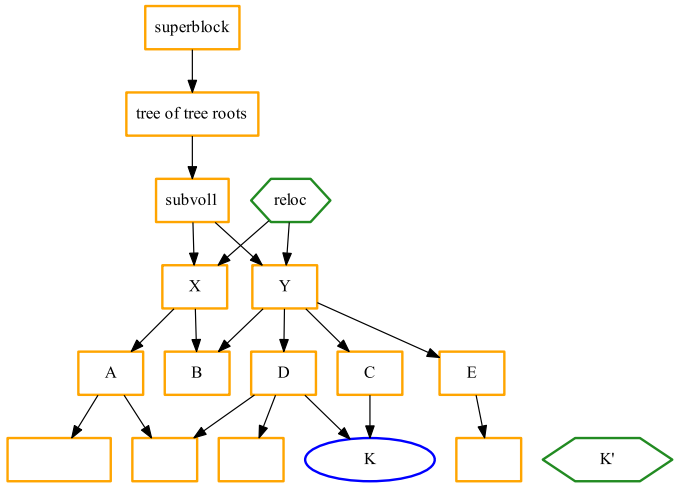


Fig. 22. Reloc trees. In this example, subvolume1 is cloned. Changes can be made to the clone.

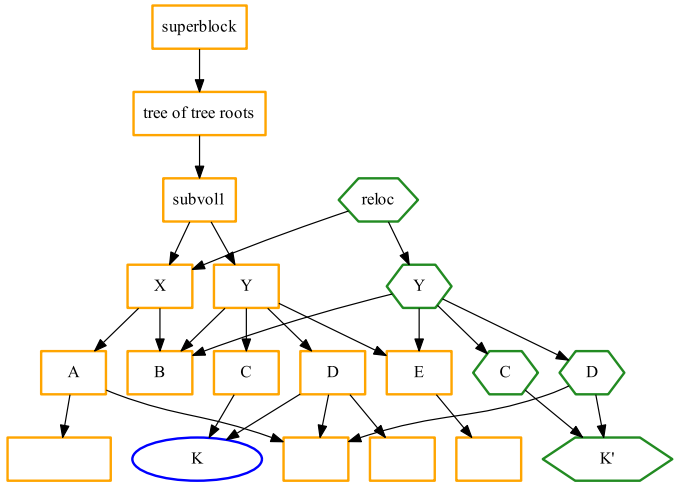


Fig. 23. Fix references in the reloc tree using COW.

(and their paths) and fix them. Finally, we drop the reloc trees—they are no longer needed. Figure 24 depicts an example.

Swapping is done using copy-on-write; nodes are never updated in place. This incurs a cost, because nodes on the path to the root need to be updated, even if they only indirectly reference the chunk. For example, node Y is COWed, only because it references nodes $\{C, D\}$, which themselves are not in the chunk, but merely reference extent K. Nodes that have no path to the chunk, such as X, do not require modification.

The new filesystem DAG is now in memory, and has the correct sharing structure. It takes the same amount of space as the original, which means that filesystem space usage stays the same. Writing out the DAG to disk can be done to contiguous extents, resulting in improved sequentiality. Once that is done, the old chunk can be discarded.

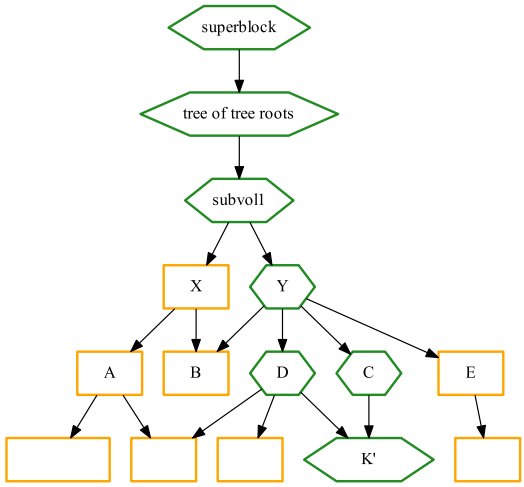


Fig. 24. Merge reloc trees with corresponding fs trees, then drop the reloc trees.

6. PERFORMANCE

There are no agreed upon standards for testing filesystem performance. While there are industry benchmarks for the NFS and CIFS protocols, they cover only a small percentage of the workloads seen in the field. At the end of the day, what matters to a user is performance for his particular application. The only realistic way to check which filesystem is the best match for a particular use case, is to try several filesystems, and see which one works best.

As we cannot cover all use cases, we chose several common benchmarks, to show that BTRFS performs comparably with its contemporaries. At the time of writing, the major Linux filesystems, aside from BTRFS, are XFS and Ext4. These are significantly more mature systems, so we do not expect to perform orders of magnitude better. Our contribution is a filesystem supporting important new features, such as snapshots and data checksums, while providing reasonable performance under most workloads.

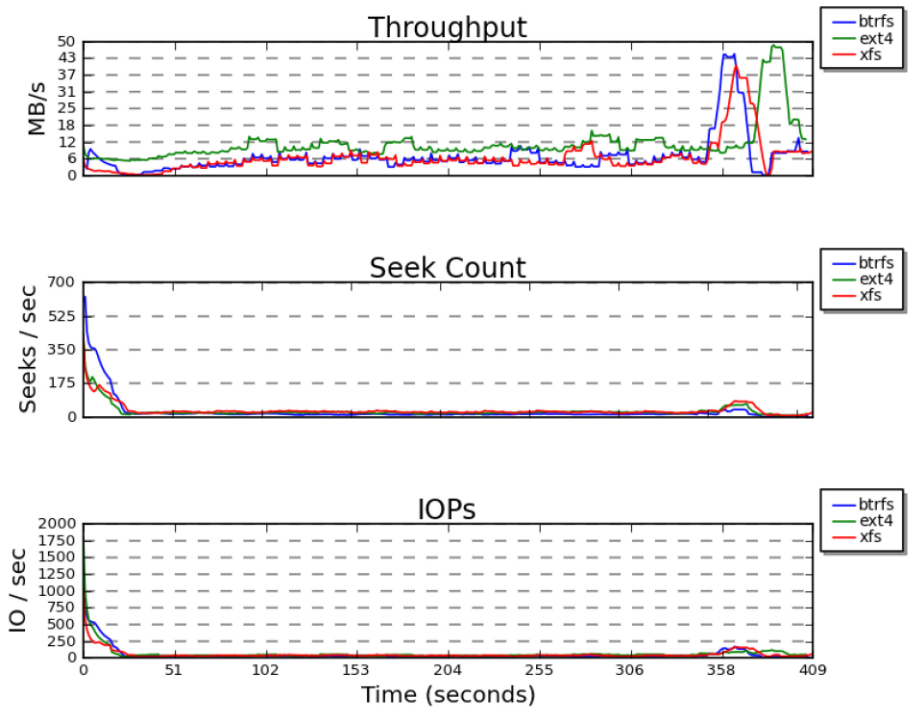
An important filesystem which we do not compare against is ZFS. Section 2.1 covers a lot of its features, and also shows that it has significant differences compared to BTRFS. While ZFS has ports to Linux, it is not a native Linux filesystem, which prevents a real apples to apples comparison.

Three storage susbsystems were examined: an SSD, a hard disk, and multiple disks in a RAID10 configuration.

6.1. Hard Disk

The tests reported here were run on a single socket 3.2 Ghz quad core x86 processor with 8 gigabytes of RAM on a single SATA drive with a 6gb/s link. The Linux version was 3.4.

The first test was a Linux kernel make, starting from a clean tree of source files. A block trace was collected, starting with the `make -j8` command. This starts eight parallel threads that perform C compilation and linking with `gcc`. Figure 25 compares throughput, seek count, and IOps between the three filesystems. Ext4 has slightly higher throughput than BTRFS and XFS, averaging a little less than twice as much throughput. All filesystems maintain about the same number of seeks per second, with BTRFS on average seeking less. The spike at the beginning of the run for BTRFS is likely to do with the initial bit of copy-on-writing that bounces between different block



	Avg Seeks/s	Avg MB/s	Avg IO/s	Run time (s)
btrfs	21.06	7.01	44.02	405.45
ext4	24.99	11.81	54.27	404.92
xfs	30.12	6.68	48.17	409.2

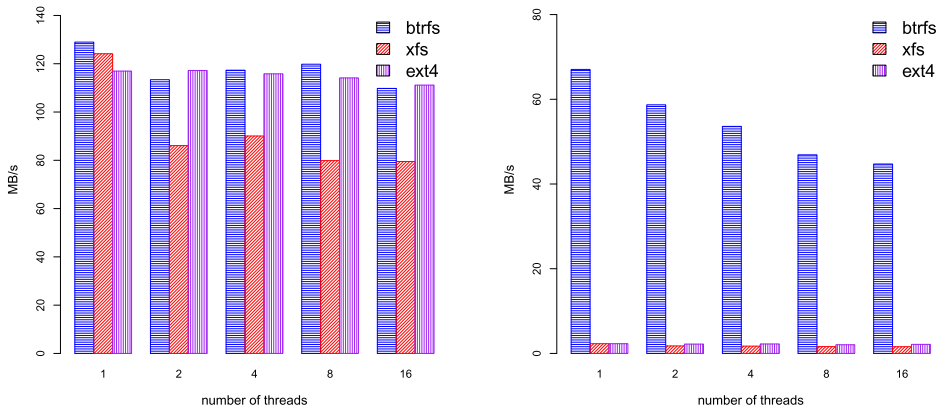
Fig. 25. A Kernel compile; all filesystems exhibit similar performance.

groups. Once additional block groups are allocated to deal with the metadata load, everything smooths out. The IO operations per second are a little closer together, but again Ext4 wins out overall. The final spike in throughput is due to writing out `vmlinux` to disk, amounting to a long sequential write. The compile times are all within a few seconds of each other. The kernel compile test tends to be a bit metadata intensive, and it is a good benchmark for an application that has a heavy metadata load. The overall picture is that the filesystems are generally on par.

The following microbenchmarks deal with write performance (see Figure 26). Tiobench writes a given size to a file with a specified number of threads. We used a 2000MB file and ran with 1, 2, 4, 8, and 16 threads. Both tests show BTRFS running the fastest overall, and in the random case, dominating the other two file systems. The random case is probably much better for BTRFS due to its write-anywhere nature, and also because we use range locking for writing instead of a global inode lock, which allows it do parallel operations much faster. Performance declines somewhat with additional threads due to contention on the shared inode mutex.

6.2. Flash Disk

Flash disks are becoming ubiquitous, replacing traditional hard disks in many modern laptops. Smartphones and embedded devices running Linux commonly use flash disks



(a) sequential (b) random

Fig. 26. TIO benchmark.

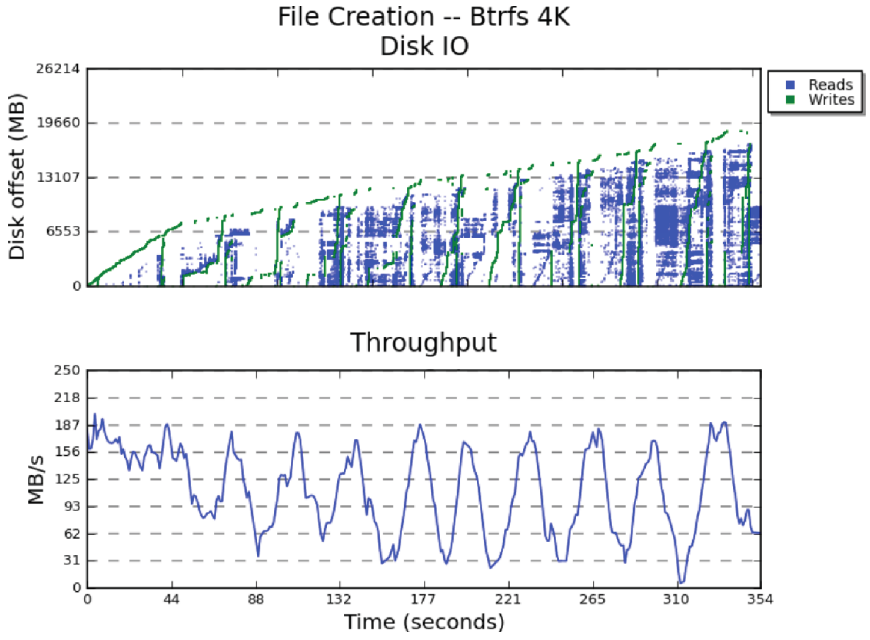


Fig. 27. Performance in the kernel 3.3. File creation rate is unsteady, throughput is a series of peaks and valleys.

as well. This has motivated an ongoing effort to optimize BTRFS for solid state drives (SSDs). Here, we describe some of this work; the code is now part of Linux kernel 3.4. The hardware used was a FusionIO^{TM} device.

Figure 27 depicts performance for the Linux 3.3 kernel, with BTRFS creating 32 million empty files. The graph was generated by Seekwatcher [Mason 2008], a tool that visualizes disk head movement. In the top graph, X axis is time, the Y axis is disk head location, reads are colored blue, and writes are colored green. The bottom graph tracks throughput. The filesystem starts empty, filling up with empty files as

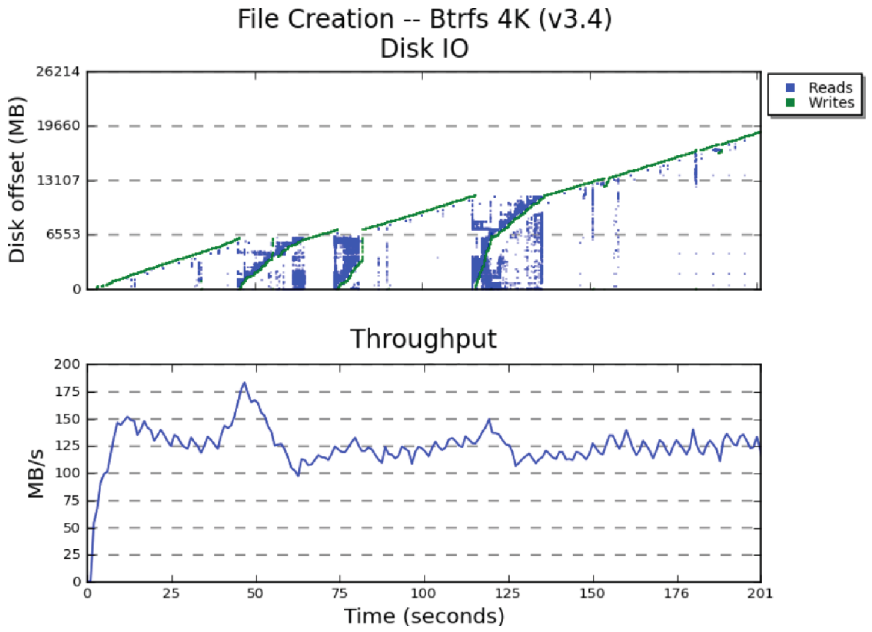


Fig. 28. Performance in the kernel 3.4, with 4KB metadata pages.

the test progresses. The pattern emerging from the graph is a continuous progression, writing new files at the end. File metadata is batched and written sequentially during checkpoints. Checkpoints take place every thirty seconds. They incur many random reads, appearing as a scattering of blue dots. The reads are required to access the free space allocation tree, a structure too big to fit in memory. The additional disk seeks slow down the system considerably, as can be seen in the throughput graph. The reason writes do not progress linearly is that checkpoints, in addition to writing new data, also free up blocks; these are subsequentally reused.

In order to improve performance, the number of reads had to be reduced. The core problem turned out to be the way the Linux virtual memory (VM) system was used. The VM assumes that allocated pages will be used for a while. BTRFS, however, uses many temporary pages due to its copy-on-write nature, where data can move around on the disk. This mismatch was causing the VM to hold many out of date pages, reducing cache effectiveness. This in turn, caused additional paging in the free space allocation tree, which was thrown out of cache due to cache pressure. The fix was for BTRFS to try harder to discard from the VM pages that have become invalid. Figure 28 shows the resulting performance improvement. The number of reads is much reduced, and they are concentrated in the checkpoint intervals. Throughput is steady at about 125MB/sec, and the rate of file creation is 150,000 files per second. Note that this optimization is general in nature, and also applies to disk workloads. The problem was not observed with HDD, because the IOps rates are much lower.

Testing showed that using larger page sizes was beneficial on flash. Figure 29 shows the effects of using a 16KB metadata page size. The rate of file creation is 170,000 files per second. Using a larger metadata page size is also important for RAID5/6 integration, as it allows putting one page on a single RAID stripe.

On the same hardware, XFS performed 115,000 files/second, and Ext4 performed 110,000 files/second. We believe this makes filesystems comparable.

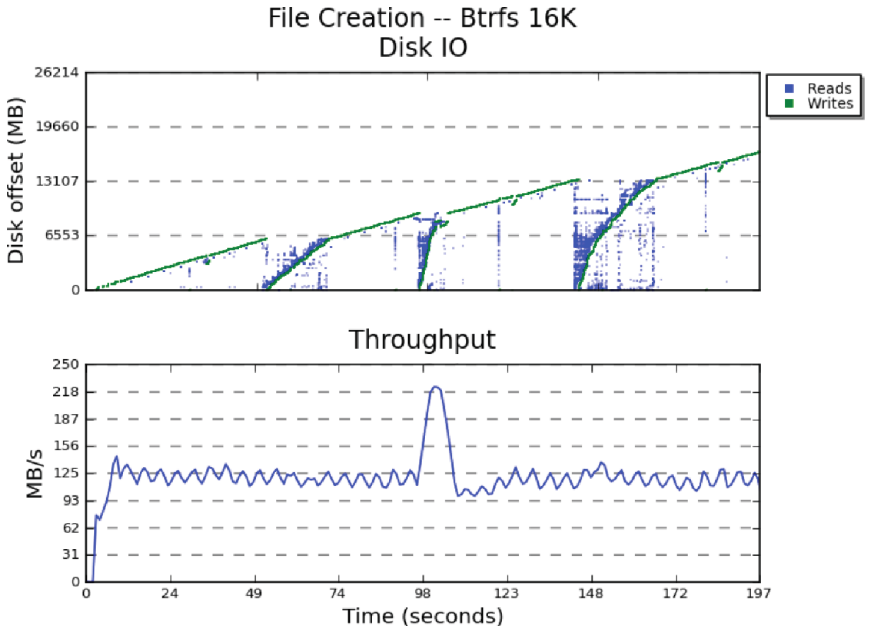


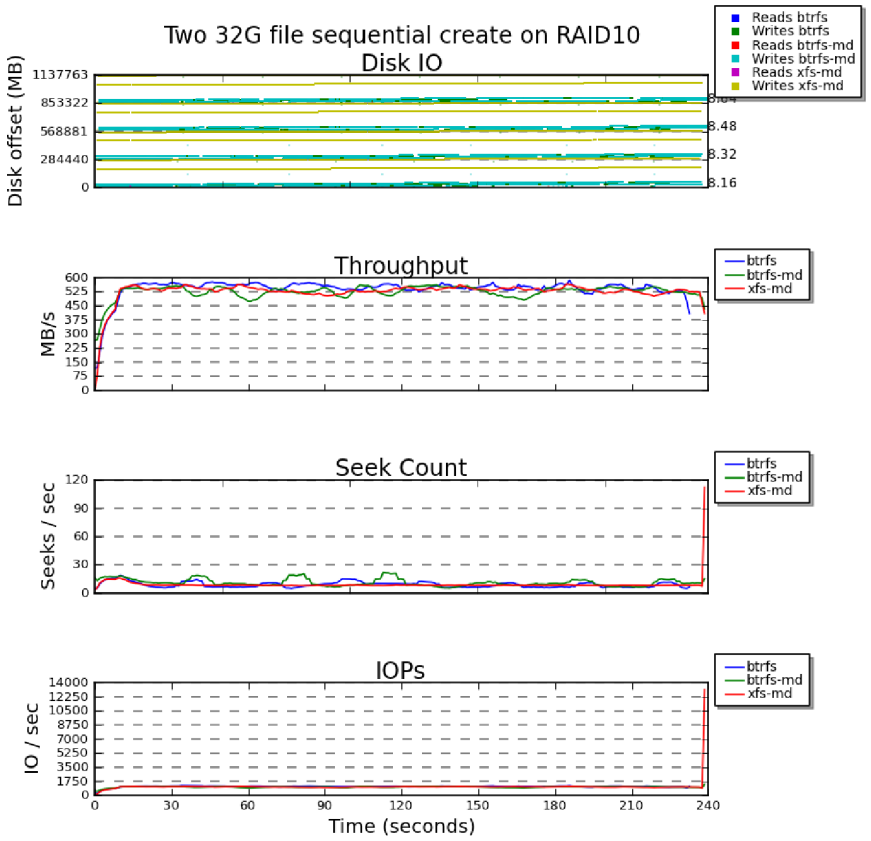
Fig. 29. Performance in kernel 3.4, with 16KB metadata pages.

6.3. Multiple Devices

The machine used in this section was a single socket AMD Phenom^{TM} II X4 840 Processor with 16GB of RAM. It has a Fedora 17 OS with a 3.6 Linux kernel. It was connected to four 3GB SATA disks each with 6Gbit/sec connectivity. A RAID10 configuration was created in three ways: (1) native BTRFS (*btrfs*), (2) BTRFS over the Linux MD layer (*btrfs-md*), and (3) XFS over MD (*xfs-md*). These three configurations were compared using two experiments.

Figure 30 shows an FIO job that creates two 32GB files, one after the other, creating a 64GB sequential write. BTRFS using its native RAID10 is slightly faster, but all of the results are very close together. We believe this is because BTRFS has lower overhead using its internal RAID10. The system always has to map its logical disk offsets to the physical disk offsets to build the block-io structure (`bio`) to send to the lower level. With native RAID, the math is done at this point, but on top of MD we incur the slight overhead of the lower level when doing the RAID math to our constructed bio.

Figure 31 shows an FIO job that reads both of the files back simultaneously and sequentially. This is probably the biggest difference between native RAID and MD, and it likely has to do with the complexity of MD's read-balancing math. Looking at the read patterns, there are sequential reads across all four disks in the case of BTRFS, but with MD there is a stair-stepping pattern between the devices. This is probably because BTRFS selects the read mirror based on the process id (PID) of the reader, so one thread will always read from the same devices in a stripe set. With two sequential readers, the pattern is that both of them read from different devices in a stripe set. This provides much less seeking overall and much higher throughput. MD applies a sophisticated algorithm that keeps track of head locations of all the disks within its array to try and make the disks seek as little as possible. However, with a multithreaded workload like this, the cleverness does not pay off. It is likely that in more complex workloads MD would be closer, and might be faster than, BTRFS.



	Avg Seeks/s	Avg MB/s	Avg IO/s	Run time (s)
btrfs	8.57	545.6	1106.7	232.37
btrfs-md	10.83	528.69	1062.6	238.21
xfs-md	8.25	530.78	1065.5	238.01

Fig. 30. RAID-10 creates two 32GB files.

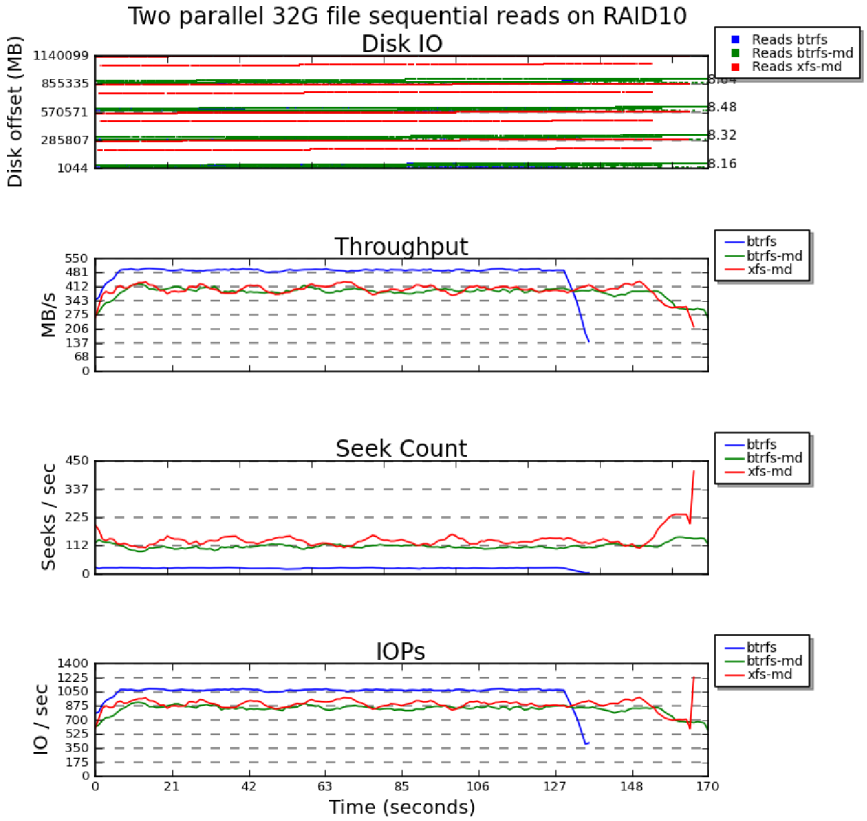
6.4. Macrobenchmarks

In this section we run several complex workloads, and see how the filesystems perform. The FileBench[3] toolkit was used, with the mail, Web, file, and OLTP personalities.

The machine used was an Intel Core^{TM} i7-2600 3.40GHz CPU. It has a single socket with eight cores, and 16GB of RAM. The memory was limited to 2GB by setting a Linux boot parameter. The OS was Ubuntu 12.10, with a 3.6.6 Linux kernel. The hard disk (*HDD*) was a SATA II, 3Gbit/sec, 7200 RPM drive. The flash disk (*SSD*) was an Intel SSD-320 Series, 300GB capacity, with R/W throughput of 270 MB/s / 205 MB/s, and R/W IOps of 39.5K/23K.

A FileBench run starts by preallocating a filesystem tree. Once that is done, the chosen workload is executed while carefully measuring performance. We looked at the operations per second (*ops/s*), and the cpu per operation (*cpu/op*) metrics. An *operation* is a benchmark step, such as reading a whole file, appending a certain

[3]http://sourceforge.net/projects/filebench



	Avg Seeks/s	Avg MB/s	Avg IO/s	Run time (s)
btrfs	21.44	479.85	1043.71	136.64
btrfs-md	110.45	385.25	837.63	170.19
xfs-md	135.94	396.87	890.58	165.13

Fig. 31. RAID-10 reads two 32GB files sequentially with two threads.

amount of data to it, and so on. Ops/s can be used to compare relative filesystem performance.

Each workload was run five times against the HDD and five times against the SSD. The execution phase of each experiment lasted ten minutes. We report on the median ops/s, median cpu/op, and the relative standard error. Generally speaking, results were fairly consistent. We did encounter a problem with OLTP, which occasionally reported very low ops/s.

Default mount options and `mkfs` programs were used. Per file system, the following options were employed.

filesystem	options
BTRFS	relatime,space_cache,rw
BTRFS w. SSD	relatime,space_cache,rw,ssd
Ext4	data=ordered,relatime,rw
XFS	attr2,noquota,relatime,rw

Workload	Avg. file size	#files	Footprint	IO size (R/W)	# threads	R/W ratio
web	32KB	350000	11GB	1MB/16KB	100	10:1
file	256KB	50000	12.5GB	1MB/16KB	50	1:2
mail	16KB	800000	12.5GB	1MB/16KB	100	1:1
OLTP	1GB	10	10GB	2KB/2KB	200+10	20:1

Fig. 32. Summary of workload parameters.

Workload	Filesystem	median ops/s	relerr(ratio)	median cpu/op
file	btrfs	432	0.00	579us
	ext4	396	0.07	356us
	xfs	231	0.00	488us
mail	btrfs	132	0.01	903us
	ext4	613	0.02	464us
	xfs	195	0.02	619us
oltp	btrfs	229	0.01	27066us
	btrfs_nocow	252	0.00	24469us
	ext4	209	0.00	30821us
	xfs	234	0.02	21327us
web	btrfs	370	0.04	796us
	ext4	859	0.00	215us
	xfs	400	0.00	274us

Fig. 33. Hard disk results.

In the OLTP case we also tested BTRFS with the `nodatacow` option. This turns off shadowing and checksums for data pages. Copy-on-write is still used for metadata.

The Web workload emulates a Web server that serves files to HTTP clients. It uses 100 threads that read entire files sequentially, as if they were Web pages. The threads, in a 1:10 ratio, append to a common file, emulating a Web log.

The file workload mimics a server that hosts home directories of multiple users. A thread emulates a user, which is assumed to access only files and directories in his home directory. Each thread performs a sequence of create, delete, append, read, write, and stat operations

The mail workload mimics an electronic mail server. It creates a flat directory structure with many small files. It then creates 100 threads that emulate e-mail operations: reading mail, composing, and deleting. This translates into many small file and metadata operations.

The OLTP workload emulates a database that performs *online transaction processing*. It is based on the IO model used by Oracle^{TM} 9i. It creates 200 data-reader threads, 10 data-writer threads, and one log thread. The readers perform small random reads, the writers perform small random writes, and the log thread does 256KB log writes. The threads synchronize using a set of semaphores, so they all work in concert.

Figure 32 summarizes the test parameters. All tests have a total on-disk footprint of 10GB or higher, at least a factor of five larger than physical RAM. This prevents the filesystem from simply caching the entire dataset in memory.

Figure 33 summarizes the hard disk results, and Figure 34 shows the corresponding SSD results. Each line in a table summarizes five experiments. Three values are presented, the median ops/s, the relative error of ops/s, and the median cpu/op.

Workload	Filesystem	median ops/s	relerr(ratio)	median cpu/op
file	btrfs	3524	0.00	834us
	ext4	3465	0.00	381us
	xfs	3151	0.00	535us
mail	btrfs	4007	0.02	929us
	ext4	11701	0.11	665us
	xfs	7616	0.03	765us
oltp	btrfs	426	0.50	17897us
	btrfs_nocow	7104	0.02	421us
	ext4	7349	0.00	385us
	xfs	9068	0.01	285us
web	btrfs	14945	0.09	549us
	ext4	17585	0.00	294us
	xfs	17828	0.00	295us

Fig. 34. SSD results.

Examining the cpu/op values for both tables, we can see that BTRFS requires more CPU cycles, although this is generally within a factor of 1.5 of Ext4 and XFS.

The HDD ops/s results are quite similar for the file and OLTP workloads. The mail server uses `fsync` quite heavily, and BTRFS suffers there, because its fsync implementation is about x4–8 slower than Ext4. This is a known problem, and improvements are planned for kernel 3.8. The Ext4 HTree data structure is a very effective index for the large flat mail directory. It is two levels deep, while the B-tree indexes used by the other filesystems are much deeper, requiring more random IO. This forms an advantage for Ext4 in the Web workload as well.

For SSD, the ops/s are reasonably close for the file and Web workloads. In the mail workload, BTRFS lags due to a less efficient fsync implementation. For OLTP, standard BTRFS does poorly, as this is a workload that does not work well with copy-on-write. Disabling COW brings results to the same ballpark as the other filesystems. In addition, OLTP had one run with very low ops/s, which caused a relative standard error of 50%.

These are just four workloads which by no means exercises all the various ways one can use a filesystem. Hopefully, they are representative of the ways most file systems are used. In most of these cases, BTRFS was in the same ballpark as its more mature counterparts. In a few cases, low fsync performance was an issue, and this problem is being addressed at the time of writing. For heavy database workloads, the nodatacow mount option will be possible per file in kernel 3.7.

7. SUMMARY

BTRFS is a relatively young Linux filesystem, working its way towards achieving default status on many Linux distributions. It is based on copy-on-write, and supports efficient snapshots and strong data integrity.

As a general purpose filesystem, BTRFS has to work well on a wide range of hardware, from enterprise servers to smartphones and embedded systems. This is a challenging task, as the hardware is diverse in terms of CPUs, disks, and memory.

This work describes the main algorithms and data layout of BTRFS. It shows the manner in which copy-on-write B-trees, reference-counting, and extents are used. It is the first to present a defragmentation algorithm for COW-based filesystems in the presence of snapshots.

ACKNOWLEDGMENTS

We would like to thank Zheng Yan, for explaining the defragmentation algorithm, and Valerie Aurora, for an illuminating LWN paper on BTRFS. Vasily Tarasov and Erez Zadok, helped us to use FileBench utility, and gave us good advice on filesystem performance testing. We are indebted to the many BTRFS developers who have been working hard since 2007 to bring this new filesystem to the point where it can be used by Linux customers and users. Finally, we would like to thank friends who reviewed the drafts, catching errors, and significantly improving quality: Gary Weiss and W.W.

REFERENCES

Bonwick, J. and Moore, B. ZFS, The last word in file systems. http://hub.opensolaris.org/bin/download/Community+Group+zfs/docs/zfslast.pdf.

Callaghan, B., Pawlowski, B., and Staubach, P. 1995. NFS Version 3 Protocol Specification. RFC 1813, IETF. June.

Chang, F., Dean, J., Ghemawat, S., Wilson, C., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber R. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Berkeley, CA, 15–15.

Comer, D. 1979. Ubiquitous B-Tree. *ACM Comput. Surv. 11*, 2, 121–137.

Dean, J. and Ghemawat, S. LevelDB. http://code.google.com/p/leveldb.

Edwards, J., Ellard, D., Everhart, C., Fair, R., Hamilton, E., Kahn, A., Kanevsky, A., Lentini, J., Prakash, A., Smith, K., and Zayas, E. 2008. FlexVol: Flexible, efficient file volume virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 129–142.

Gailly, J. L. and Adler, M. ZLIB. en.wikipedia.org/wiki/Zlib.

Heizer, I., Leach, P., and Perry, D. 1996. Common Internet File System Protocol (CIFS/1.0). Draft draft-heizer-cifs-v1-spec-00.txt, IETF.

Hellwig, C. 2009. XFS: The big storage file system for Linux. In *Usenix Login Magazine*.

Henson, V., Ahrens, M., and Bonwick, J. 2003. Automatic performance tuning in the Zettabyte File System. In *File and Storage Technologies (FAST), Work in Progress Report*. USENIX Association, Berkeley.

Hitz, D., Lau, J., and Malcolm, M. 1994. File system design for an NFS file server appliance. In *USENIX*. USENIX Association, Berkeley, CA.

Konishi, R., Sato, K., and Amagai, Y. NILFS. www.nilfs.org.

Macko, P., Seltzer, M., and Smith, K. 2010. Tracking back references in a Write-Anywhere file system. In *Proceedings of 8th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Berkeley, CA.

Mason, C. 2007. BTRFS. http://en.wikipedia.org/wiki/Btrfs.

Mason, C. 2008. Seekwatcher. http://oss.oracle.com/~mason/seekwatcher.

Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. 2007. The new Ext4 Filesystem: Current status and future plans. In *Proceedings of Linux Symposium.*

O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica 33*, 4, 351–385.

Patterson, D., Gibson, G., and Katz, R. 1988. A Case for redundant arrays of inexpensive disks (RAID). *SIGMOD 17,* 3, 109–116.

Reed, I. S. and Solomon, G. 1960. Polynomial codes over certain finite fields. *J. Society Indus. Appl. Math. 8*, 300–304.

Reiser, H. 2001. ReiserFS. http://http://en.wikipedia.org/wiki/ReiserFS.

Ren, K. and Gibson, G. 2012. TABLEFS: Embedding a NOSQL database inside the local file system. Tech. rep. CMU-PDL-12-103.

Rodeh, O. 2006a. B-trees, shadowing, and clones. Tech. rep. H-248, IBM.

Rodeh, O. 2006b. B-trees, shadowing, and range-operations. Tech. rep. H-248, IBM.

Rodeh, O. 2008. B-trees, shadowing, and clones. *ACM Trans. Storage 3,* 4.

Rodeh, O. 2010. Deferred reference counters for Copy-On-Write B-trees. Tech. rep. rj10464, IBM.

Rosenblum, M. and Ousterhout, J. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

Sears, R. and Ramakrishnan, R. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 217–228.

Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. 2000. NFS version 4 Protocol. RFC 3010, IETF.

Shetty, P., Spillane, R., Malpani, R., Andrews, B., Seyster, J., and Zadok, E. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*.

Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. 1996. Scalability in the XFS File System. In *Proceedings of USENIX Annual Technical Conference*.

Received July 2012; revised November 2012, March 2013; accepted March 2013