RJ10501 (ALM1207-004) July 9, 2012
Computer Science

IBM Research Report

BTRFS: The Linux B-tree Filesystem

Ohad Rodeh
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Josef Bacik, Chris Mason
FusionIO

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

BTRFS: The Linux B-tree Filesystem

Ohad Rodeh — IBM
Josef Bacik — FusionIO
Chris Mason — FusionIO

Abstract

BTRFS is a Linux filesystem, headed towards mainline default status. 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.

A Linux filesystem is 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 harddisks. It is also expected to be able to use an array of different sized disks; posing challenges to the RAID and striping mechanisms.

This paper 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.

1 Introduction

BTRFS is an open source filesystem that has seen extensive development since its inception in 2007. It is jointly developed by Fujitsu^{TM}, Fusion-IO^{TM}, Intel^{TM}, Oracle^{TM}, Red Hat^{TM}, Strato^{TM}, SUSE^{TM}, 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. Multi-device support
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 serve as the default Linux filesystem; 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 [14] 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 paper 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.

The approach taken in this paper 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 [1].

This paper 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 [17], XFS [5], and BTRFS [3]. In the class of copy-on-write filesystems, two important contemporary systems are ZFS [18], and WAFL [7, 11]. 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 C. Mason. He combined ideas from ReiserFS [8], with COW friendly b-trees suggested by O. Rodeh [14], 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)* 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 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. As it is an in-place replacement for Ext3, older filesystems can seamlessly be upgraded. Ext4 is an overwrite based filesystem, that 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, and snapshots are not supported.

XFS 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 that 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.

ZFS 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. ZFS includes volume-manager functionality, protects data and metadata with checksums, and supports space-efficient snapshots. RAID5/6 is supported with RAID-Z, which has the interesting feature of always writing full stripes. Space is managed with variable sized blocks,

which are powers of two; all space for a single file is allocated with one block size. In terms of features, ZFS is generally similar to BTRFS, however, the internal structures are quite different. For example, BTRFS manages space in extents, where ZFS uses blocks. BTRFS uses b-trees, where ZFS uses traditional indirect blocks.

WAFL is the filesystem used in the NetApp^{TM} commercial file server; development started in the early 1990's. It is a copy-on-write filesystem that is especially suited for NFS [2, 16] and CIFS [9] 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 client receives an acknowledgment from the server. 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 mature and feature rich filesystem.

ReiserFS [8] 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 per 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.

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 standard Linux page size.

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: [14, 12, 13, 15]. The main idea is to use standard b+-tree construction [6], but (1) employ a top-down update procedure, (2) remove leaf-chaining, (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 colored yellow, and COWed pages are colored green.

Figure 1(a) shows an initial tree with two levels. Figure 1(b) shows an insert of 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.

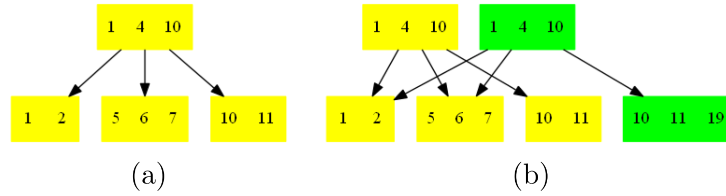


Figure 1: (a) A basic b-tree (b) Inserting key 19, and creating a path of modified pages.

In order to remove a key, copy-on-write is used. Remove operations do not modify pages in place. For example, Figure 2 shows how key 6 is removed from a tree. Modifications are written off to the side, creating a new version of the tree.

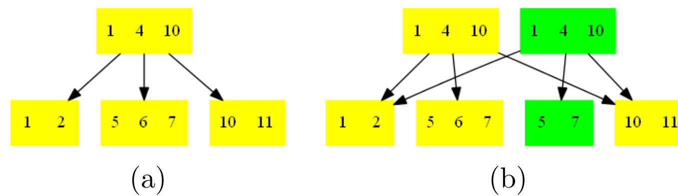
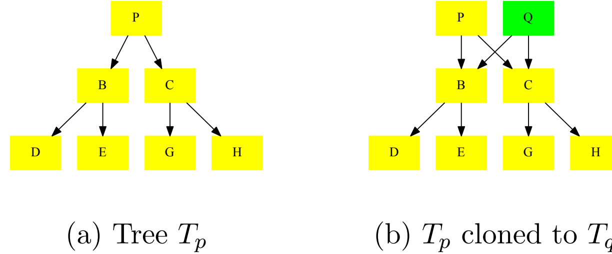


Figure 2: (a) A basic tree (b) Deleting key 6.

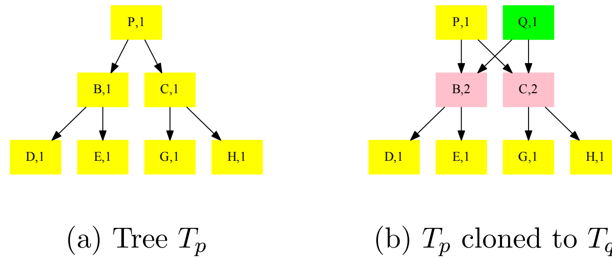
In order to clone a tree, its root node is copied, and all the child pointers are duplicated. For example, Figure 3 shows a tree T_p, that is cloned to tree T_q. Tree nodes are denoted by symbols. As modifications will be applied to T_q, sharing will be lost between the trees, and each tree will have its own view of the data.



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

Figure 3: Cloning tree T_p. A new root Q is created, initially pointing to the same blocks as the original root P. As modifications will be applied, the trees will diverge.

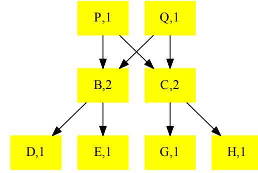
Since tree nodes are reachable from multiple roots, garbage collection is needed for space reclamation. In practice, file systems are directed acyclic graphs (DAGs). There are multiple trees with shared nodes, but there are no cycles. Therefore, reference-counters (*ref-counts*) can and are used to track how many pointers there are to tree nodes. Once the counter reaches zero, a block can be reused. 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. For example, Figure 4 shows the clone example with a ref-count indication. The convention is that pink nodes are unchanged except for their ref-count.



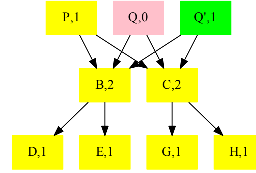
(a) Tree T_p (b) T_p cloned to T_q

Figure 4: Cloning tree T_p. A new root Q is created, initially pointing to the same blocks as the original root P. The ref-counts for the immediate children are incremented. The grandchildren remain untouched.

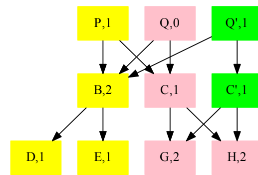
Figure 5 shows an example of an insert-key into leaf H, tree q. The nodes on the path from Q to H are $\{Q, C, H\}$. They are all modified and COWed.



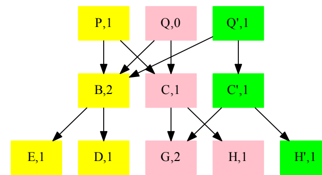
(a) Initial trees, T_p and T_q



(b) Shadow Q



(c) shadow C



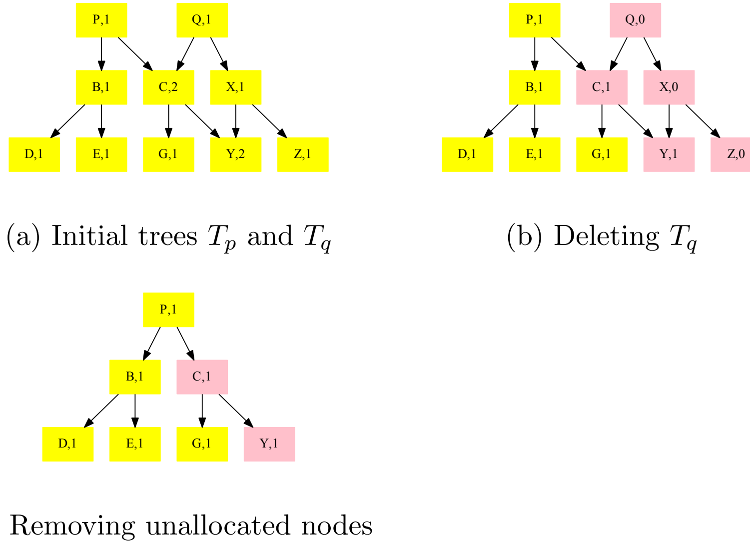
(d) shadow H

Figure 5: 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.

Figure 6 shows an example of a tree delete. The algorithm used is a recursive tree traversal, starting at the root. For each node N:

- 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.



(a) Initial trees T_p and T_q

(b) Deleting T_q

(c) Removing unallocated nodes

Figure 6: 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.

3.2 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, etc. A key describes an object address using the structure:

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

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

```
struct item {
    struct 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 variable 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 Table 7 shows a leaf with three items $\{I_0, I_1, I_2\}$ and three corresponding data elements $\{D_2, D_1, D_0\}$.

Figure 7: A leaf node with three items. The items are fixed size, but the data elements are variable sized.

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 8 shows a more detailed schematic of a leaf node.

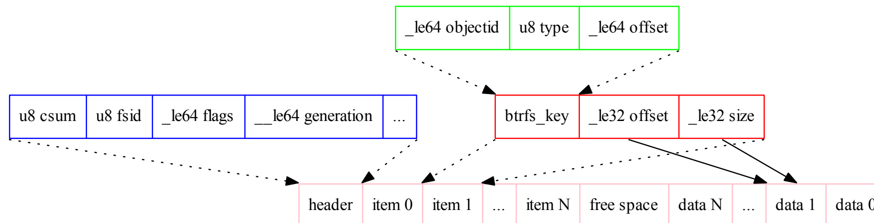


Figure 8: A detailed look at a generic leaf node holding keys and items.

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. The inode structure is relatively small, and will not contain embedded file data or extended attribute data. 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 (explained below) for the extent and a `[disk block, disk num blocks]` pair to record the area of disk corresponding to the file. Extents also store the logical offset and the number of blocks used by this extent record into the extent on disk. This allows performing a rewrite into the middle of an extent without having to read the old file data first. For example writing 10MB into extent `0 - 64MB` can cause the creation of three different extents: `0 - 10MB, 10-20MB, 20-64MB`.

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.

A directory holds an array of `dir_item` elements. A dir_item maps a filename (string) to a 64bit object_id. The directory also contains two indexes, one used for lookup, the other for iteration. The lookup index is an array with pairs `[dir_item key, filename 64bit hash]`, it is used for satisfying path lookups. The iteration index is an array with pairs `[dir_item key, 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.

3.3 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:

Sub-volumes: store user visible files and directories. Each sub-volume is implemented by a separate tree. Sub-volumes can be snapshotted and cloned, creating additional b-trees. The roots of all sub-volumes 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 ex-

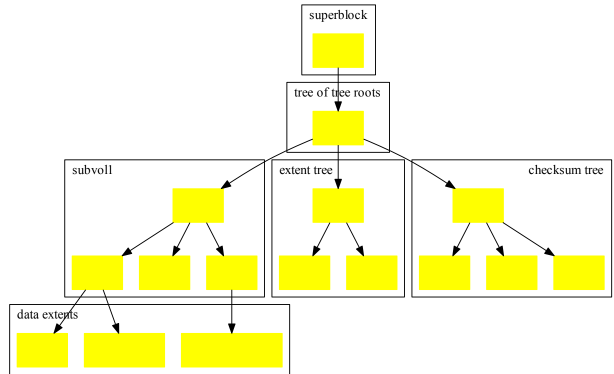
tent are recorded in the extent item. This allows moving an extent if needed, or recovering from a damaged disk block. Taken as a whole, back-references multiply the number of filesystem disk pointers by two. For each forward pointer, there is exactly one back-pointer. See more details on this tree below.

Checksum tree: holds a `checksum item` per allocated extent. The item contains a list of checksums per page in the extent.

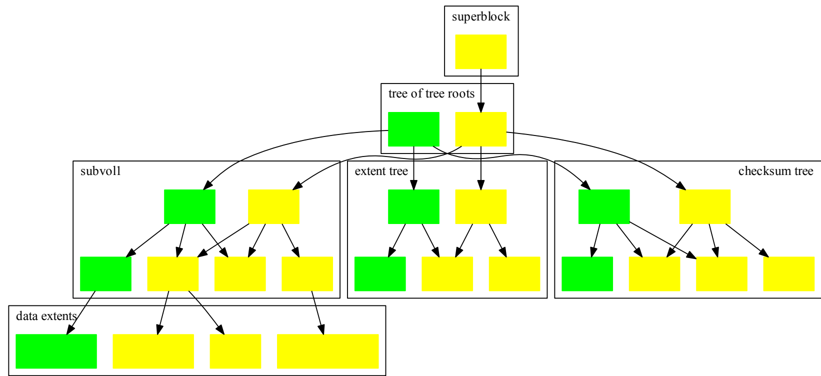
Chunk and device trees: indirection layer for handling physical devices. Allows mirroring/striping and RAID. Section 4 shows how multiple device support is implemented using these trees.

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

For example, Figure 9(a) shows a high-level view of the structure of a particular filesystem. The reloc and chunk trees are omitted for simplicity. Figure 9(b) shows the changes that occur after the user wrote to the filesystem.



(a)



(b)

Figure 9: (a) A filesystem forest. (b) The changes that occur after modification; modified pages are colored green.

Modifying user-visible files and directories causes page and extent updates. These ripple up the sub-volume tree until 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 top most 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.

Sub-volume trees can be snapshoted and cloned, and they are therefore ref-counted. All other trees keep meta-data 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 co-access. 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 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, this allows detection of phantom or misplaced writes on the media. Generation numbers and checksums serve together to verify on disk block content.

3.4 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` that 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:

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

Figure 10: 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

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 logical, not physical. 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.

3.5 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.

3.6 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 requires 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 *device-mapper* (DMs) subsystems that manage storage devices. For example, `LVM` and `mdadm`. These are software modules whose 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. They support mirroring, striping, and RAID5/6. However, 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 be about 1% 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 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. Table 1 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}. Table 2 shows a case where one disk is larger than the other two.

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}

Table 1: 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}

Table 2: One large disk, and two small disks, in a RAID1 configuration.

For striping, groups of n chunks are used, where each physical chunk is on a different disk. For example, Table 3 shows stripe width of four ($n = 4$), with four disks, and three logical chunks.

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}

Table 3: Striping with four disks, stripe width is $n = 4$. Three logical chunks are each made up of four physical chunks.

At the time of writing, RAID levels 0,1, and 10 are supported. In addition, there is experimental code by Intel^{TM} that supports RAID5/6. The core idea in higher RAID levels is to use chunk groups with Reed-Solomon [10] parity relationships. For example, Table 4 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.

	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}

Table 4: 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}.

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.

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 that tries to spread allocations evenly on all available disks, even as the device population changes. For example, in Table 5(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 Table 5(b). The goal of the balancing code is to reach the state shown in Table 5(c), where data is spread evenly on all three disks, and each disk holds $\frac{1}{3}$ of the raw data.

(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}

Table 5: Device addition. Initially (a), there are two disks. In state (b), another disk is added, it 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.

When a device is removed, the situation is reversed. From a $\frac{1}{3}$ ratio (as in Table 5(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

At the time of writing, the defragmentation problem is addressed in two separate ways. 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. In many cases, this simple algorithm is sufficient. In some cases, a more sophisticated approach, that maintains sharing, is needed.

When shrinking a filesystem, or evicting data from a disk, 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:

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 11 shows a simple example. One data extent, colored light blue, needs to be moved.

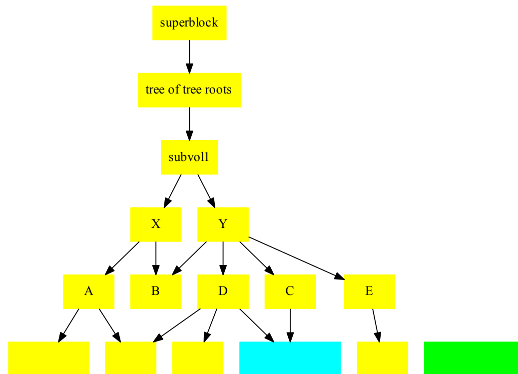


Figure 11: Do a range lookup in the extent tree, find all the extents located in the chunk. Copy all the live extents to new locations.

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 12.

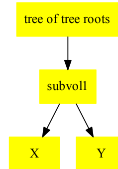


Figure 12: Upper level nodes stored in a backref_cache.

A list of sub-volume trees that reference the chunk from the backref_cache is calculated; these trees are subsequently cloned, see Figure 13. This operation has two effects: (1) it freezes in time the state of the sub-volumes (2) it allows making off-to-the-side modifications to the sub-volumes while affording the user continued operation.

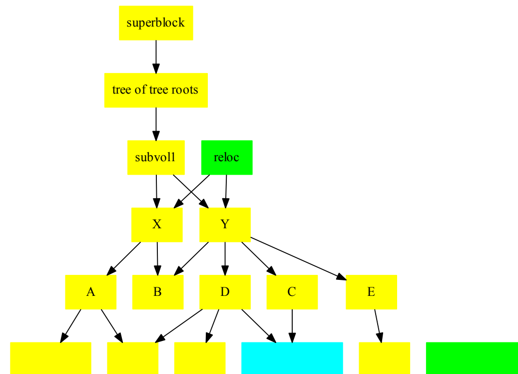


Figure 13: Reloc trees. In this example, sub-volume1 is cloned. Changes can be made to the clone.

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 14.

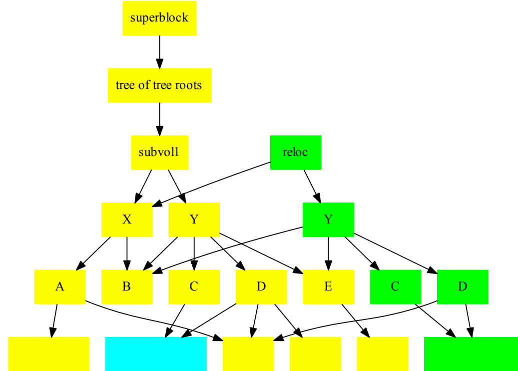


Figure 14: Fix references in the reloc tree using COW.

The last step is to merge the reloc trees with the originals. We traverse the trees. We find sub-trees that are modified in the reloc tree but where corresponding parts in the fs tree are not modified. These sub-trees in the reloc tree are swapped with their older counterparts from the fs tree. The end result is new fs-trees. Finally, we drop the reloc trees, they are no longer needed. Figure 15 depicts an example.

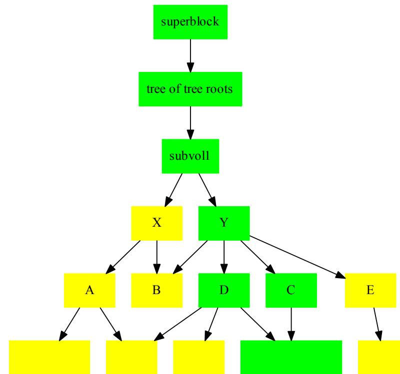


Figure 15: Merge reloc trees with corresponding fs trees, then drop the reloc trees.

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 onto contiguous extents resulting in improved sequentiality. Once that is done, the old chunk can be discarded.

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 percent 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.

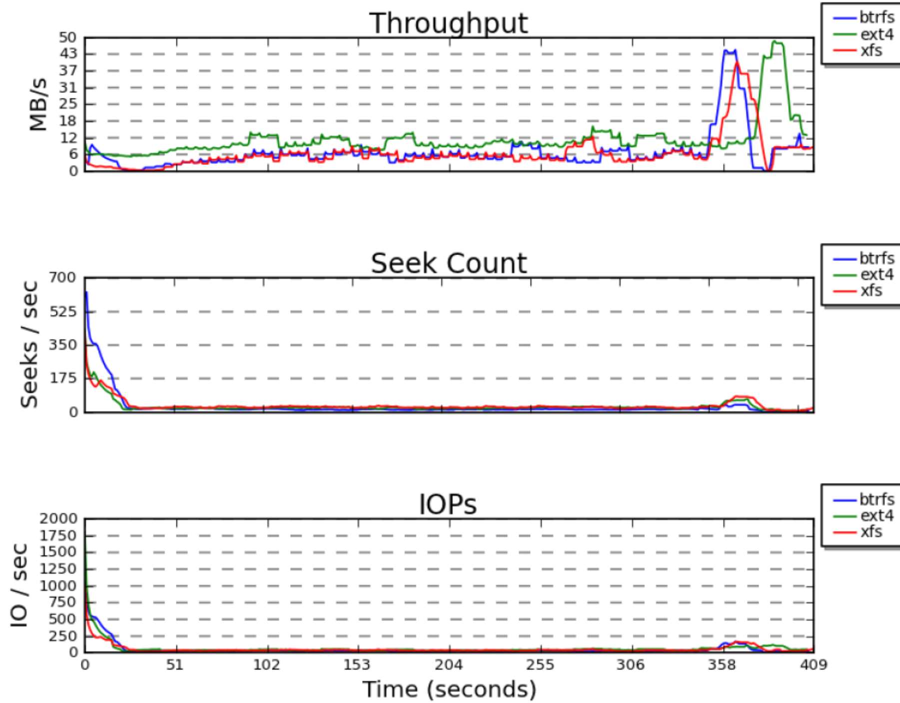
As we cannot cover all use cases, we chose several common benchmarks, to show that BTRFS performs comparably with its contemporaries. At the time of writing, the major Linux filesystems, aside from BTRFS, are XFS and Ext4. These are significantly more mature systems, and we do not expect to perform orders 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.

Two storage configurations were chosen: a hard disk, and an SSD.

6.1 Hard disk

All of the hard disk tests where 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 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 16 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 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 groups. Once additional block groups are allocated to deal with the meta-data load, everything smooths out. The IO operations per second are a little closer together, but again Ext4 wins out overall. The compile times are all within a few seconds of each other. The kernel compile test tends to be a bit meta-data intensive, and it is a good benchmark for an application that has a heavy meta-data load. The overall picture is that the filesystems are generally on par.



	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

Figure 16: A Kernel compile, all filesystems exhibit similar performance.

The FFSB test attempts to mimic a mail server. It creates 500000 files in 1000 directories all ranging from 1KB to 1MB in size, weighted more towards 32KB size or less. Once it creates the files, it spends 300 seconds doing either creates, deletes or reading entire files, all with a block size of 4KB. The test weighs reading higher than creating, and creating higher than deleting in order to try and represent how a mail server would work. The test can be modified to use any number of threads, for simplicity, we used one thread. FFSB measures throughput and transactions per second, shown in 17, and 18.

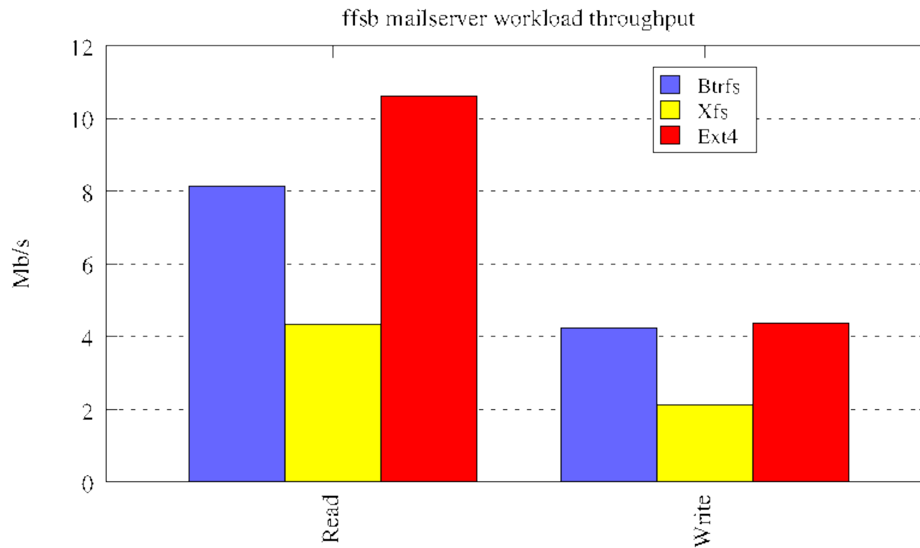


Figure 17: Throughput during an FFSB mail server run. XFS uses the least bandwidth, Ext4 uses the most, and BTRFS is in the middle.

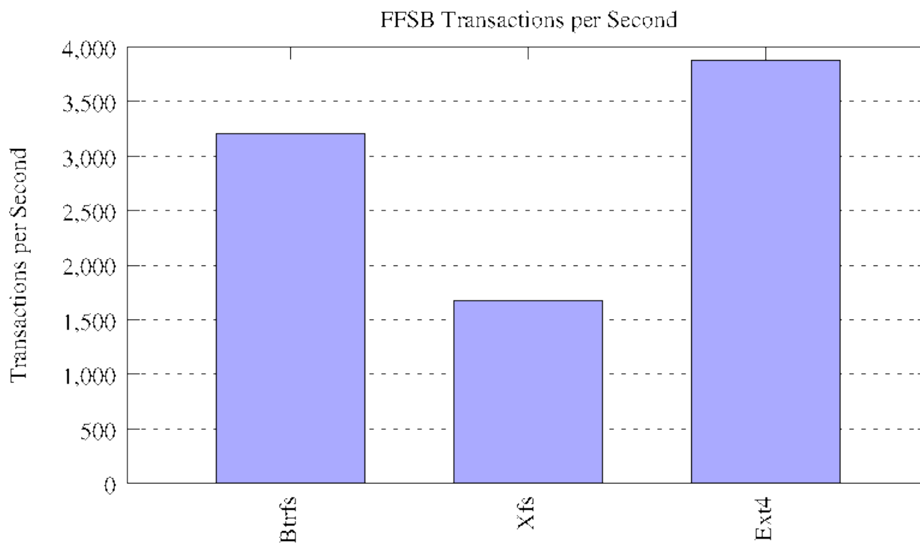


Figure 18: Transactions per second in the FFSB. BTRFS shows comparable performance.

This workload favors Ext4, with BTRFS trailing slightly behind and XFS performing at half the speed of BTRFS.

The final test deals with write performance. 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 makes it do parallel operations much faster. Performance declines somewhat with additional threads due to contention on the shared inode mutex.

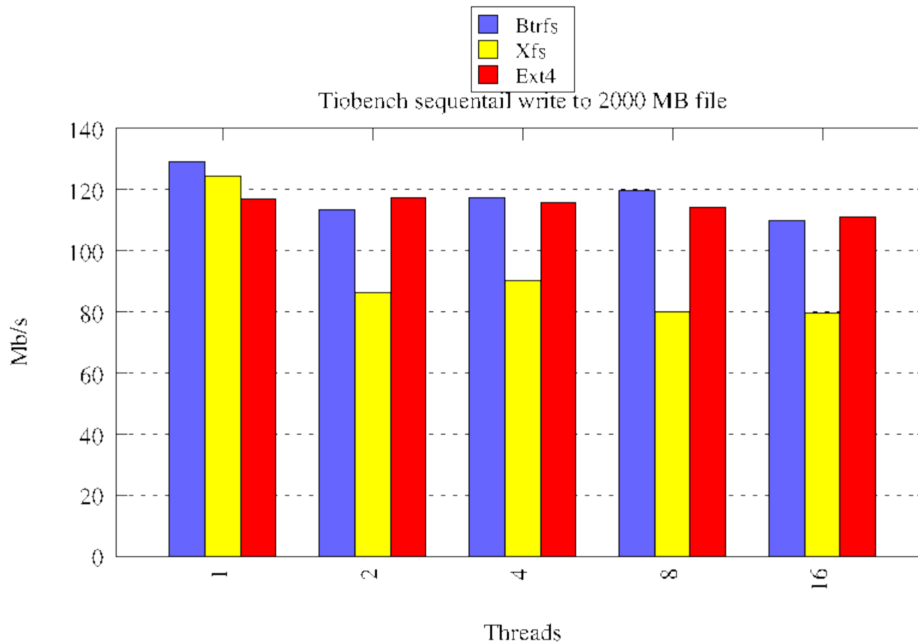


Figure 19: TIO benchmark, sequential.

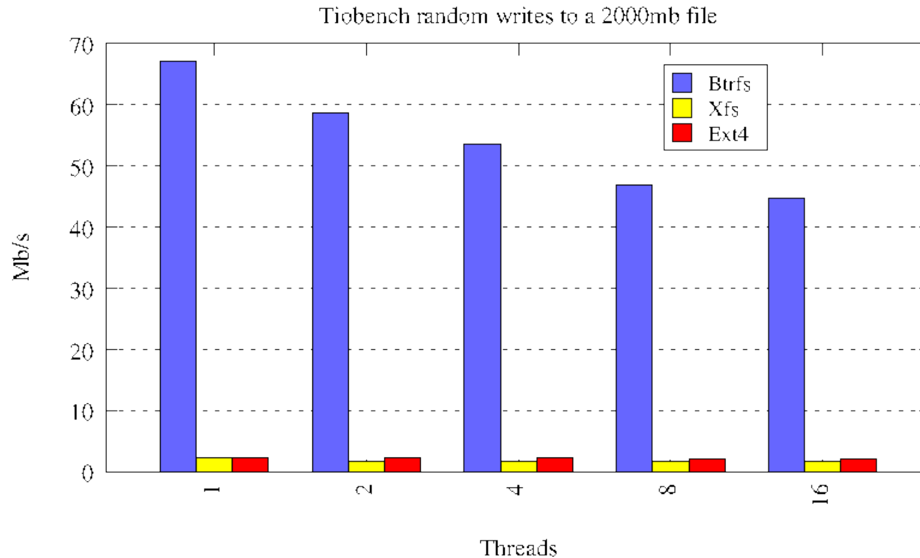


Figure 20: TIO benchmark, random

These are just three tests and by no means exercise all the various ways one can use a filesystem. Hopefully, they are representative of the ways most file systems are used. In all these cases, BTRFS was in the same ballpark as its more mature counterparts.

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 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 21 depicts performance for the Linux 3.3 kernel, with BTRFS creating 32 million empty files. The graph was generated by Seekwatcher [4], 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 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.

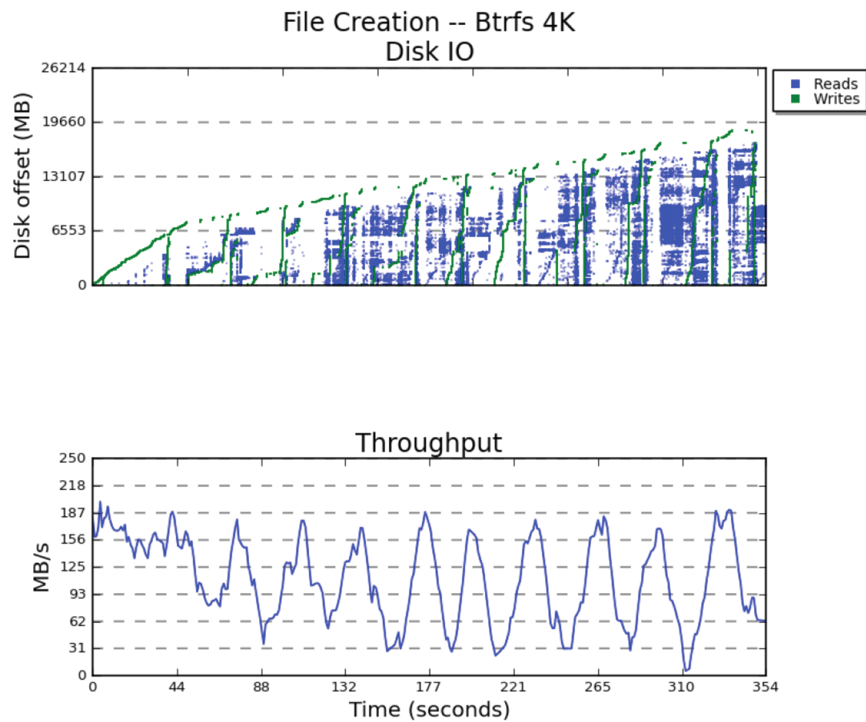


Figure 21: Performance in the kernel 3.3. File creation rate is unsteady, throughput is a series of peaks and valleys.

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 22 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.

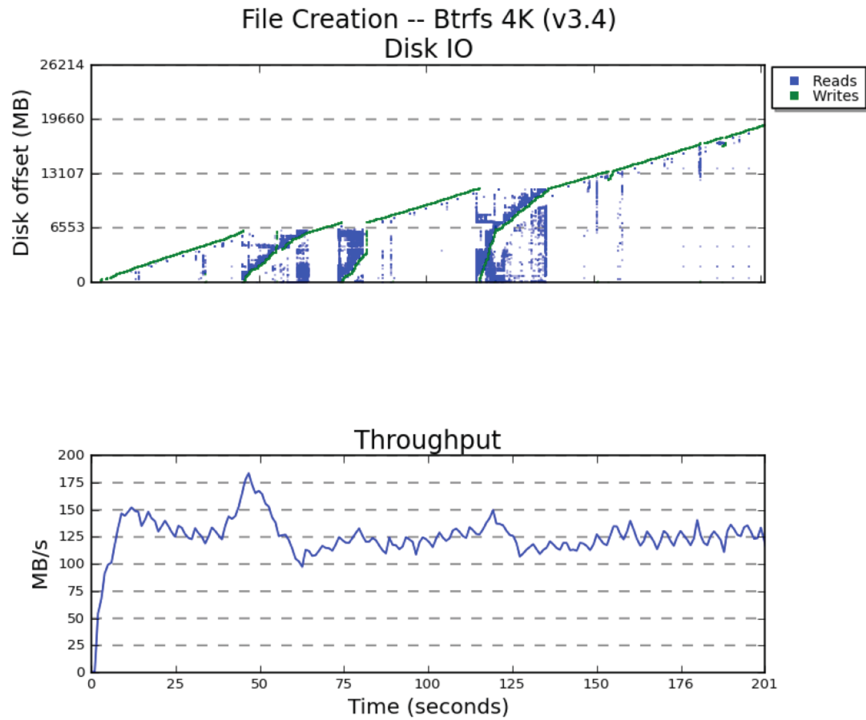


Figure 22: Performance in the kernel 3.4, with 4KB metadata pages.

Testing showed that using larger page sizes was beneficial on flash. Figure 23 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.

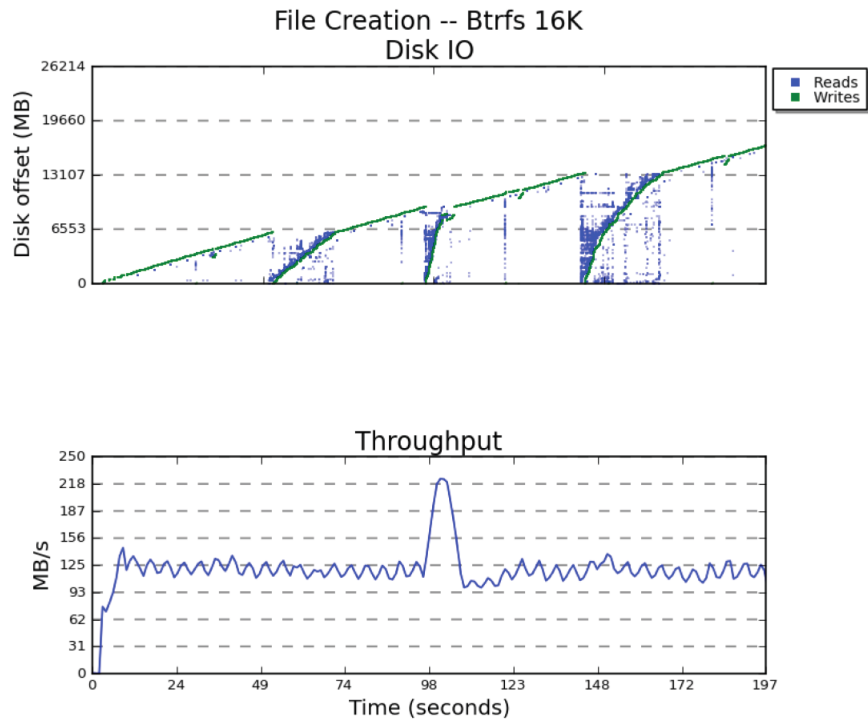


Figure 23: Performance in the kernel 3.4, with 16KB metadata pages.

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

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.

8 Acknowledgments

We would like to thank Zheng Yan, for explaining the defragmentation algorithm, and Valerie Aurora, for an illuminating LWN paper on BTRFS.

Thanks is also due 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 Valentin and W.W..

References

[1] BTRFS mailing list. http://www.mail-archive.com/-linux-btrfs@vger.kernel.org/index.html.

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

[3] C. Mason. BTRFS. http://en.wikipedia.org/wiki/Btrfs.

[4] C. Mason. Seekwatcher, 2008.

[5] D. Chinner. XFS. http://en.wikipedia.org/wiki/XFS.

[6] D. Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[7] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *USENIX*, 1994.

[8] H. Reiser. ReiserFS. http://http://en.wikipedia.org/wiki/ReiserFS.

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

[10] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[11] J. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. Smith, and E. Zayas. Flexvol: flexible, efficient file volume virtualization in wafl. In *USENIX Annual Technical Conference*, pages 129–142, Berkeley, CA, USA, 2008. USENIX Association.

[12] O. Rodeh. B-trees, shadowing, and clones. Technical Report H-248, IBM, November 2006.

[13] O. Rodeh. B-trees, shadowing, and range-operations. Technical Report H-248, IBM, November 2006.

[14] O. Rodeh. B-trees, Shadowing, and Clones. *ACM Transactions on Storage*, Feb 2008.

[15] O. Rodeh. Deferred Reference Counters for Copy-On-Write B-trees. Technical Report rj10464, IBM, 2010.

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

[17] T. Tso. Fourth Extended Filesystem (EXT4). http://en.wikipedia.org/wiki/Ext4.

[18] V. Henson, M. Ahrens, and J. Bonwick. Automatic Performance Tuning in the Zettabyte File System. In *File and Storage Technologies (FAST), work in progress report*, 2003.