

# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ  
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INFORMATION SYSTEMS

## ONLINE DEDUPLICATION FOR BTRFS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. MARTIN KŘÍŽEK

BRNO 2013



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
BRNO UNIVERSITY OF TECHNOLOGY



**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**  
**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INFORMATION SYSTEMS

## **ONLINE DEDUPLIKACE PRO BTRFS**

ONLINE DEDUPLICATION FOR BTRFS

**DIPLOMOVÁ PRÁCE**

MASTER'S THESIS

**AUTOR PRÁCE**

AUTHOR

**Bc. MARTIN KŘÍŽEK**

**VEDOUcí PRÁCE**

SUPERVISOR

**Ing. TOMÁŠ KAŠPÁREK**

BRNO 2013

## Abstrakt

Btrfs je copy-on-write linuxový souborový systém, který obsahuje vlastního správce oddílů a podporuje efektivní operace se snapshoty. Online deduplikace dat je metoda odstranění duplicitních bloků dat ještě předtím než jsou zapsány na disk. Tato funkcionality šetří místo na uložišti avšak může v jistých případech znamenat pokles výkonu při zápisu. Hlavním využitím jsou soubory obrazů disků virtuálních strojů. Takové soubory obsahují z velké části stejná data (stejný operační systém) a tedy tyto části lze zapsat na disk pouze jednou. Tato diplomová práce se zabývá návrhem a implementací online deduplikace pro souborový systém Btrfs.

## Abstract

Btrfs is a copy-on-write Linux filesystem that has its own built-in volume management and supports efficient snapshotting. Online data deduplication is a technique of eliminating duplicate blocks before they are written out to disk. This feature saves storage space, however, might decrease performance in some cases. The most notable use case for this feature is virtual machine image files. Most of the blocks are of the same content (same operating system) in these files and thus those blocks might be written out to disk just once. This thesis deals with design and implementation of such feature for the Btrfs filesystem.

## Klíčová slova

linux, kernel, btrfs, souborový systém, online deduplikace

## Keywords

linux, kernel, btrfs, file system, online deduplication

## Citace

Martin Křížek: Online deduplication for Btrfs, diplomová práce, Brno, FIT VUT v Brně, 2013

# Online deduplication for Btrfs

## Prohlášení

Hereby I declare that I have written this master's thesis on my own under the supervision of Ing. Tomáš Kašpárek. I have acknowledged all sources I have used while writing this thesis.

.....  
Martin Křížek  
22.5.2013

## Poděkování

I would like to thank Ing. Tomáš Kašpárek for supervising this thesis and guiding my work. Also, I would like to thank Ing. Lukáš Czerner from Red Hat Czech, s.r.o. for his professional comments on my work. Finally, I would like to thank my parents for their support throughout my studies.

© Martin Křížek, 2013.

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Filesystems</b>	<b>4</b>
2.1	Ext4 . . . . .	4
2.2	XFS . . . . .	4
2.3	ReiserFS . . . . .	5
2.4	ZFS . . . . .	5
2.5	Summary . . . . .	5
<b>3</b>	<b>Copy-on-write Friendly B-trees</b>	<b>6</b>
3.1	Overview . . . . .	6
3.2	Algorithms . . . . .	7
<b>4</b>	<b>Btrfs</b>	<b>9</b>
4.1	Overview . . . . .	9
4.2	Design . . . . .	10
4.3	Using the Filesystem . . . . .	15
<b>5</b>	<b>Data Deduplication</b>	<b>18</b>
5.1	Overview . . . . .	18
5.2	Types of Deduplication . . . . .	18
5.3	Use Cases . . . . .	20
5.4	Finding Duplicates . . . . .	20
5.5	ZFS Deduplication . . . . .	20
5.6	Btrfs Offline Deduplication . . . . .	21
<b>6</b>	<b>Analysis</b>	<b>23</b>
6.1	High-Level Feature Overview . . . . .	23
6.2	Controls . . . . .	23
6.3	Limitations . . . . .	24
6.4	Level of Deduplication . . . . .	24
6.5	Initiation of Deduplication . . . . .	24
6.6	Deduplication Block . . . . .	25
6.7	Deduplication Table . . . . .	25
6.8	Hash Algorithm . . . . .	26
6.9	Referencing a Duplicate . . . . .	26

<b>7</b>	<b>Implementation</b>	<b>28</b>
7.1	Initiating Deduplication . . . . .	28
7.2	In-Memory Hashes . . . . .	28
7.3	Deduplication Table . . . . .	29
7.4	Writing Deduplication Metadata . . . . .	30
7.5	Actual Deduplication . . . . .	32
7.6	Controlling Deduplication . . . . .	32
<b>8</b>	<b>Recent work</b>	<b>34</b>
8.1	Implementation . . . . .	34
8.2	Features . . . . .	35
8.3	Comparison . . . . .	35
8.4	Summary . . . . .	36
<b>9</b>	<b>Testing and Results</b>	<b>37</b>
9.1	Test Case 1: Disk Usage . . . . .	37
9.2	Test case 2: Read – Metadata Fragmentation . . . . .	38
9.3	Test case 3: Read – Data Fragmentation . . . . .	38
9.4	Test case 4: Write Unique Block . . . . .	38
9.5	Test case 5: Write Duplicate Block . . . . .	38
9.6	Testing environment . . . . .	38
9.7	Testing . . . . .	39
9.8	Development Testing . . . . .	39
9.9	Summary . . . . .	40
<b>10</b>	<b>Future Work</b>	<b>41</b>
<b>11</b>	<b>Conclusion</b>	<b>43</b>
<b>A</b>	<b>CD content</b>	<b>47</b>
<b>B</b>	<b>Manual</b>	<b>48</b>
B.1	Installation . . . . .	48
B.2	Usage . . . . .	49

# Chapter 1

## Introduction

Data deduplication is a technique of eliminating duplicate data and thus saves space in storage. Consequently, saving storage space leads to decreasing network bandwidth. There are several approaches for data deduplication. Duplicates can be disposed before they are actually written to storage (online or immediate deduplication) or they can be eliminated from the storage after being written (offline or post-process deduplication). Then, there are several levels of how data are deduplicated. Data can be deduplicated on bytes, blocks or files. Each approach has its advantages and disadvantages that will be presented in the thesis. The technique might be implemented right in a filesystem, in an application (an e-mail server deduplicates e-mail attachments e.g.) or possibly in network devices.

Btrfs is relatively new linux filesystem based on copy-on-write friendly b-trees that has its own volume manager and supports efficient snapshotting. While the filesystem already supports quite a few key features, there are still several features to be implemented. One of them is online deduplication. This advanced feature saves storage space in use cases like backup servers, mail server or virtual machine images. Note that while online deduplication is advantageous in the previously mentioned cases, it can decrease performance in other use cases and so it is not by any means a feature that every user would benefit from. Designing a data structure that would hold information about filesystem blocks and that could allow a fast lookup for a duplicate is the main challenge in designing such feature. This thesis deals with design and implementation of online deduplication for Btrfs. The rest of the thesis is organized as follows.

*Chapter 2* shortly describes the most commonly used general purpose linux filesystems. Copy-on-write b-trees that Btrfs is based on are presented in *chapter 3*. *Chapter 4* deals with the Btrfs filesystem itself, describing its internal data structures, how these structures are used and usage of the filesystem from the user stand-point concludes the chapter. Different data deduplication techniques are presented in *chapter 5*. *Chapter 6* deals with design of the online deduplication feature for the Btrfs filesystems. Implementation challenges, data structures and algorithms used to complete the new feature are covered in the *chapter 7*. The recent work that has been done in the time of writing the thesis is presented in the *chapter 8*. Next, in follow-up *Chapter 9*, the previously mentioned chapter is completed with benchmarks of implementation. *Chapter 10* briefly outlines possibilities for improvements of online deduplication for Btrfs in the future. Finally, *chapter 11* summarizes the thesis.

# Chapter 2

## Filesystems

In this chapter, the most commonly used general purpose filesystems will be briefly presented. Because of the character of the thesis, only Linux filesystems will be focused on, with ZFS being an exception as it is common to compare Btrfs with it. This chapter is not meant to be an in-depth description of filesystems, instead, its purpose is to mention interesting features that might be related to the Btrfs filesystem.

### 2.1 Ext4

Ext4 [1] (*The Fourth Extended Filesystem*) is the default Linux filesystem that was forked from the previous version, ext3, in order to improve performance and deal with its limitations. The stable version was included in the Linux kernel [2] in 2008.

Direct/indirect block addressing, that ext2/3 use, is replaced with an extent tree as a block mapping scheme. The new block mapping scheme improves performance when accessing large files. The problem with direct/indirect block mapping is that for a file, each block is mapped separately. With the extent tree, only a single extent might be mapped increasing data access time (blocks does not need to be tracked separately from direct/indirect links).

Another feature worth mentioning is delayed allocation. As the name suggests, data allocation is delayed as much as possible (at the actual on-disk write time) in order to limit data fragmentation. Delaying data allocation works well with extents. If data are being written into a file while still being in the cache, delayed allocation results in allocating extent that is optimized for actual size of data on disk.

A journal is used to ensure filesystem consistency. Snapshots are not supported in this filesystem, one would need to use LVM (that in turn use Device Mapper) to create snapshots of ext4 partitions. Users of ext3 can upgrade their partitions to ext4.

### 2.2 XFS

XFS [3] [4] [5] was developed by Silicon Graphics, Inc. and introduced in 1994 in Irix, SGI's Unix-based operating system. It was later ported to Linux. Like ext4, XFS uses delayed allocation and is extend-based filesystem. It uses B-tree of extents to manage free space. A journal is used to ensure filesystem consistency. XFS aims at high scalability being capable of having filesystems as large as 9 exabytes. Snapshots are not supported by XFS but one

can use volume manager XLV on Irix or LVM on Linux in order to create snapshots. XFS partitions cannot be shrunk.

## 2.3 ReiserFS

ReiserFS [6] [7] was developed by Hans Reiser and employees of his company named Namesys. It uses metadata only journal to keep filesystem consistent and was actually the first journaled filesystem to be included in the Linux kernel. ReiserFS filesystems can be resized online. To increase performance, a tail packing is implemented that packs last partial blocks of different files into one block. Like ext4 and XFS, delayed allocation is present in ReiserFS (although introduced in its successor, *Reiser4*).

## 2.4 ZFS

ZFS [8] was developed by Sun Microsystems, Inc. and introduced in Solaris in 2005. It is a COW<sup>1</sup> (see chapter 3) filesystem that manages storage space with variable sized blocks. Users can set block size according to their workloads. Snapshotting (read-only) and cloning (writable) are supported, being advantage of COW. ZFS presents a pooled storage model that eliminates a need for additional volume manager like LVM from Linux. Both data and metadata are checksummed in ZFS. RAID 0,1,5,6 and RAID-Z, which always writes full stripes and thus eliminates „write holes“ (failure of a disk between writing data and updating parity), are supported. Compression is supported in ZFS as well. Compression uses variable block sizes, if data after compression fits into smaller block size, the smaller block size is used. ZFS volumes can be used as swap devices and can be compressed. Although ZFS is similar and commonly compared to Btrfs in terms of features, their internals are different. There is a native port of ZFS for the linux kernel, the port is in its early stage of development in the time of writing the thesis.

## 2.5 Summary

In this section, feature summary [9] of previously presented filesystems is shown below:

	<b>ext4</b>	<b>XFS</b>	<b>ReiserFS</b>	<b>Reiser 4</b>	<b>ZFS</b>
<b>Extents</b>	Yes	Yes	No	Yes	No
<b>Metadata checksums</b>	Yes	No	No	No	Yes
<b>Resizable volumes</b>	Online	Online <sup>a</sup>	Online	Online <sup>b</sup>	Online <sup>c</sup>
<b>Snapshotting</b>	No	No	No	No	Yes
<b>Transparent compression</b>	No	No	No	Yes	Yes
<b>Encryption</b>	Yes	No	No	Yes	Yes
<b>COW</b>	No	No	No	Yes	Yes

---

<sup>a</sup>cannot be shrunk

<sup>b</sup>can only be shrunk offline

<sup>c</sup>cannot be shrunk

---

<sup>1</sup>copy-on-write

## Chapter 3

# Copy-on-write Friendly B-trees

This chapter shortly describes copy-on-write friendly b-trees [10] that Btrfs is based on. First, an overview of the issues that led to introduce a new type of b-tree is presented. Algorithms that are performed on these b-trees conclude the chapter.

### 3.1 Overview

When using the copy-on-write technique, data (extent, page) are not overwritten but written to a new location on disk instead. They are loaded from disk into memory, modified and written to a new location which prevents them from being corrupted by power failure for instance. If a failure occurs, old data are used instead.

B-tree is a data structure used in some filesystems to store files and directories. The advantage, compared to classic indirect blocks, is logarithmic time search, insert and remove operations. Shadowing (copy-on-write) is a technique to make sure that persistent data structures are atomically updated. It means that once a node (in case of a tree data structure) is updated, it is stored at a different location than the original.

With shadowing applied, inserting a key into a tree's node results in updating all its ancestors. Shadowing propagates up to the tree's root. Since the classic b-tree (see figure 3.1) has *leaf-chaining* (adjacent leafs are linked together making it easy for rebalancing and range lookups), the whole tree needs to be updated (shadowed).

The original nodes that were previously shadowed are later deallocated. Shadowing facilitates snapshotting. The original nodes are preserved to form a snapshot. Nodes are then shared between snapshots, until any of them changes.

Concurrency-wise, when a node is updated, only the node needs to be locked. With shadowing applied, the whole path from the node up to the root needs to be locked.

Another problem when applying shadowing to the classic b-trees is modifying a single path. When a node is updated and needs to be rebalanced between its adjacent nodes, it should be rebalanced with a node that is already in the path so minimum of the nodes are shadowed. In the tree from figure 3.1, if the node  $Z$  is to be updated and is full, items should be moved to the node  $A$  instead of the node  $B$  so the path  $X \rightarrow Y \rightarrow Z$  remains.

The classic b-trees are built bottom-up. Once data in a node overflow (or less than the bottom limit), the node is split (merged) and changes propagate to the next level.

Having described the above issues with the classic b-trees, the following changes are made to them to support shadowing:

1. to use top-bottom b-tree,
2. to remove leaf-chaining,
3. to use lazy reference-counting for free space map.

The basic algorithms based on copy-on-write friendly b-trees are presented in the remainder of the chapter.

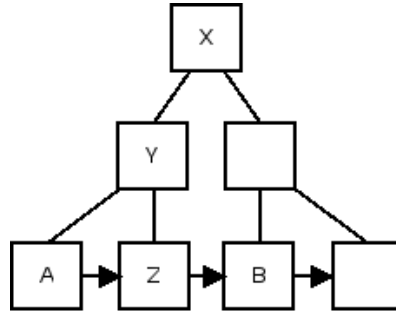


Figure 3.1: The classic B-tree with leaf-chaining

## 3.2 Algorithms

### 3.2.1 Creating a Tree

In order to create a tree, a root node is created. The root node have to contain zero to  $2b + 1$  items, while other nodes have to consist of  $b$  to  $2b + 1$  items. When creating a clone  $T'$  of a tree  $T$ , reference counts (*ref-counts*) of only immediate children nodes of the root of the tree  $T$  are incremented (this is called *lazy reference counting*).

### 3.2.2 Deleting a Tree

To delete a tree, a post-order traversal is performed on it and all the nodes are deallocated (deleted). However, while traversing the tree the ref-count of nodes must be taken into an account (the tree might be a clone and thus shares nodes with another tree(s)) with the following:

1. if the ref-count of a node is higher than one, decrement the ref-count and stop downward traversal; the node is shared between more trees
2. if the ref-count of a node is one, continue downward traversal and on the way back, delete the node

### 3.2.3 Inserting a Key

While searching a leaf for a key to be inserted all full nodes along the path are split. This ensures that at most the leaf is split. Lock-coupling (locking a child before unlocking the parent) is applied during the traversal. Special care must be taken while shadowing nodes along the path:

1. if the ref-count of a node is one, no special action is needed, the node is shadowed
2. if the ref-count of a node is greater than one, it is copied to an alternate location, the original node's ref-count is decremented by one and the newly copied node's ref-count is set to one

Figure 3.2 demonstrates a key insertion.

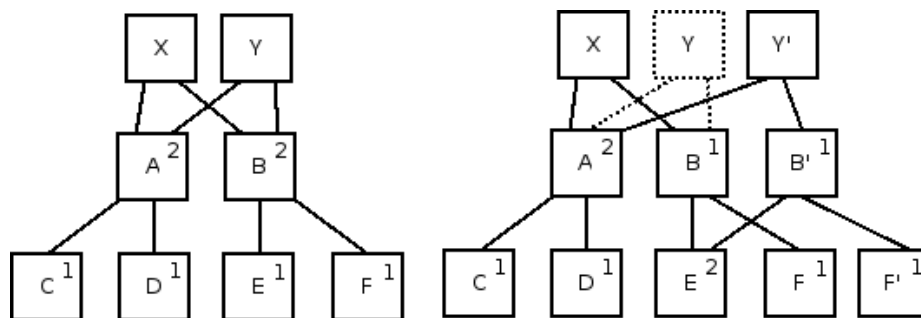


Figure 3.2: A B-tree before and after a key insertion into the leaf F. The number in a node denotes ref-count. Nodes marked with an apostrophe were shadowed.

### 3.2.4 Removing a Key

While searching a tree for a leaf containing the key to be removed from, all nodes along the path that contain a minimal number of keys are merged. This ensure that merging a leaf node will only affects its immediate ancestor, at worst. Lock-coupling is applied here as well. Special care must be taken while shadowing nodes along the path, in the exact same way as it is done with the key insertion.

# Chapter 4

## Btrfs

This chapter focuses solely on the Btrfs [11] [12] [13] [14] filesystem. First of all, the filesystem will be shortly introduced. Next, description of the design of Btrfs will follow. Finally, a set of commands that control Btrfs filesystem in the userspace will be shown to conclude the chapter.

### 4.1 Overview

Btrfs is a copy-on-write filesystem originally developed by Chris Mason in Oracle, now being developed jointly by companies like Red Hat, Oracle, Intel, Fujitsu, SUSE and others. It aims at scaling to the large number of storage and being able to detect, repair and tolerate errors in the data stored in those storages. Its goal is to become the next major Linux filesystem.

Btrfs uses COW friendly b-trees, the whole filesystem is a forest of b-trees. Copy-on-write updating method ensures that the filesystem is consistent at all times. Btrfs stores data in extents. Checksums are maintained for all metadata and data, ensuring integrity of data stored in a filesystem. Thanks to its design, writable snapshots are very efficient (a new tree root is created, pointing to the original tree's blocks, see 4.2).

Another main feature of Btrfs is multi-device support. Btrfs has its own device management and thus does not need LVM to manage devices. As of time of writing the thesis, levels 0, 1 and 10 of RAID are supported. Devices can be added and removed online using the device management.

Btrfs handles small files efficiently as they are stored along with metadata making their lookup fast. Transparent data compression is supported. Data are compressed in separate kernel threads. LZO and zlib compress algorithms can be used for compression. Ext3 and ext4 filesystems can be converted to Btrfs, and even rolled back (this is however not recommended for production use) – the original ext3/4 filesystem is preserved since Btrfs modifies data in COW fashion. Scrubbing (scanning through the filesystem looking for corrupted blocks and replacing them with good copies, if available) is done online.

## 4.2 Design

### 4.2.1 Data Structures

In Btrfs, everything (an i-node, file data, a directory entry) is an item in a COW friendly b-tree. Using one data structure, b-trees, for every object in the filesystem makes the search for objects follow one common path. There are three data structures that the Btrfs' b-tree knows of: block headers, keys and items (figure 4.1 show the structures in more detail<sup>1</sup>). Block headers has fixed size and contains information like generation number, checksum, flags, filesystem id, etc. The block header is defined as follows:

```
struct btrfs_header {
    u8  csum[32];
    u8  fsid[16];
    __le64  blocknr;
    __le64  flags;

    u8  chunk_tree_uid[16];
    __le64  generation;
    __le64  owner;
    __le32  nritems;
    u8  level;
}
```

The key structure is defined below:

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

Each object has its object id which makes up the most significant bits of the key allowing items of one object to be logically grouped together in the b-tree. Object ids are allocated dynamically on object's creation. The type field describes the kind of data held by an item. The offset field describes the data held in an extent.

The item structure is defined below:

```
struct item {
    struct key key; __le32  offset; __le32  size;
}
```

As shown above, the item structure holds a key identifying the object stored in data belonging to the item. The offset field tells where the item data are stored within the leaf and the size field has obvious meaning.

Non-leaf nodes contains [key, block-pointer] pairs while leaf nodes hold [item, data] pairs. Item data are variable sized but items are fixed sized. A leaf node is organized in a way that items are placed in the beginning (after the block header) of the leaf and data are stored at the end and so they grow towards each other, leaving a free space in the middle:

*[header] [item0, item1, item2] [free space] [data2, data1, data0]*

---

<sup>1</sup>The picture is inspired by [https://btrfs.wiki.kernel.org/index.php/Btrfs\\_design](https://btrfs.wiki.kernel.org/index.php/Btrfs_design)

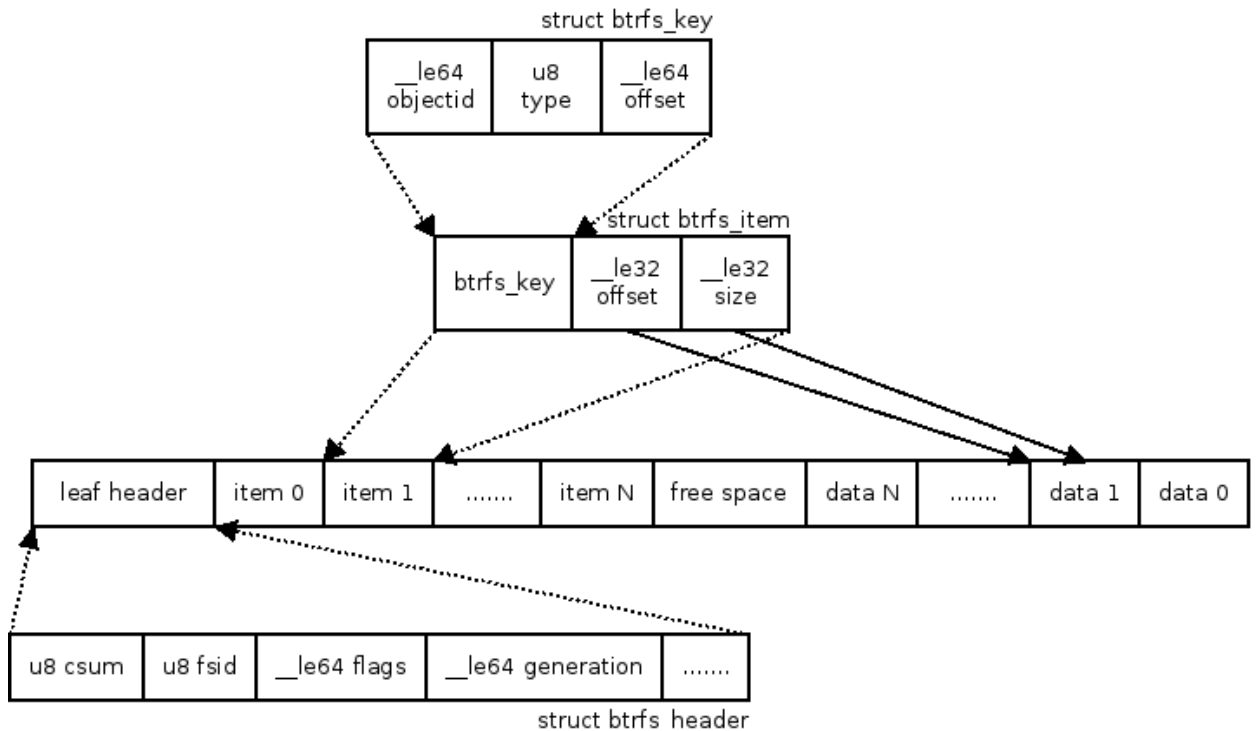


Figure 4.1: The b-tree leaf structure.

### Sample Search Code

To show how the above works, function `find_next_chunk` (`volumes.c`) is shown below.

Function `find_next_chunk` finds next chunk to be used by a new logical disk chunk in a tree. Basically, it finds the highest key in the chunk tree and returns a key that is one larger.

The structure of keys in the chunk trees is as follows: the `objectid` field holds logical byte offset of the chunk, the `offset` field holds number of bytes in the chunk and `type` has value of `BTRFS_CHUNK_ITEM_KEY`.

```
static int find_next_chunk(struct btrfs_root *root, u64 *objectid)
{
    struct btrfs_path *path;
    int ret;
    struct btrfs_key key;
    struct btrfs_key found_key;

    path = btrfs_alloc_path();
```

A path is allocated to hold a result of a search. After the search is done, `path->nodes` should contain all nodes that were gone through while searching, `path->nodes[0]` being the leaf node. `path->slots` holds slots that were chosen to direct the search through the tree, `path->slots[0]` holds a slot of the actual item in the leaf node that was found.

```
    key.objectid = (u64)-1;
    key.offset = (u64)-1;
```

```
key.type = BTRFS_CHUNK_ITEM_KEY;
```

A key for the search is set to the highest possible value that BTRFS\_CHUNK\_ITEM\_KEY may contain.

```
ret = btrfs_search_slot(NULL, root, &key, path, 0, 0);  
  
if (ret < 0)  
    goto error;
```

```
BUG_ON(ret == 0);
```

Since it is not possible to have a chunk starting at logical offset  $2^{64}$ , *btrfs\_search\_slot* is expected to return 1 (none found). Searching is a read-only operation so there is no need to set up a transaction.

```
ret = btrfs_previous_item(root, path, 0, BTRFS_CHUNK_ITEM_KEY);
```

*btrfs\_previous\_item* goes backward the tree and tries to find previous BTRFS\_CHUNK\_ITEM\_KEY. Non-zero return value means that nothing was found.

```
if (ret) {  
    *objectid = 0;  
} else {  
    btrfs_item_key_to_cpu(path->nodes[0], &found_key,  
                          path->slots[0]);  
    *objectid = found_key.objectid + found_key.offset;  
}
```

*btrfs\_item\_key\_to\_cpu* copies the item out of an extent buffer into *found\_key* and next object id is return in *objectid*.

```
ret = 0;  
error:  
    btrfs_free_path(path);  
    return ret;  
}
```

### 4.2.2 A Forest

The Btrfs filesystem consists of a forest of b-trees. Superblock, pointing at the tree of tree roots is located at the fixed position. Trees that make up the forest are as follows:

#### Reloc Tree

Shrinking, rebalancing and defragmentation operations need to relocate extents. Reloc tree serves as a temporary place to store affected metadata during these operations. This is to preserve references.

#### Checksum Tree

Checksum tree holds a checksum item per allocated extent. The item holds a list of checksums per block in the extent. The offset field in the key indicates where checksummed data starts on disk. The data is checksummed after any compression or encryption is done so it corresponds to the data sent to disk.

## Chunk and Device Tree

Each device is divided into chunks. Chunks are 1 gigabyte in size for data and 256 megabytes for metadata by default. A chunk tree keeps mapping from logical to physical chunks. A device tree maintains the exact opposite, maps physical chunks to logical. Extents are addressed logically, making it possible to move chunks between physical devices without a need to go through the tree and fix extent references. The types of keys stored in the chunk tree are *DEV\_ITEM* (that contain information on all the block devices in the filesystem, the *offset* field holds *dev\_id*) and *CHUNK\_ITEM* (the *offset* field holds a virtual address of location where the chunk starts).

## Extent Allocation Tree

The extent allocation tree [15] maintains extents (range of on disk contiguous blocks) that are in use in extent items. The extent allocation tree serves as an on-disk free-space map. The tree maintains reference counts of extents and back-references to a file or a tree that use particular extent. A reference can also point just to a part of the extent.

Assume there is a file *thesis.tex* its data are stored in an extent that has on-disk location 500kb - 564kb. The file is then cloned and later on a range of 16kb is written in the middle of the *clone*. The *clone* now consists of the following extents: 500kb - 520kb, 710kb - 726kb and 536kb - 564kb. The second extent is allocated further on disk. Now, there are three references to the 500kb - 564kb extent (*thesis.tex* points to the whole extent and the *clone* points to first 20kb and last 8kb).

Once there are more references that fit in a single leaf, the leaf is split. The *extent\_item* contains [disk block, disk num blocks] pairs to determine where data are located on disk; disk offset and length of the extent. The item also holds generation number of the extent.

There are two types of extents: block group extents and data extents. The block group extents indicate which virtual addresses are valid; whether there is a physical storage underlying them. Block groups extent key is following:

$$(start, BLOCK\_GROUP\_ITEM, length)$$

Data extents are allocated from block group extents. Content of data extents are actual filesystem data and metadata. Data extent key is as follows:

$$(start, EXTENT\_ITEM, length)$$

## FS Tree

FS trees store files and directories, all normal filesystem metadata. There is one FS tree root for each subvolume or snapshot. Snapshots share blocks between trees. Roots are indexed by the tree of tree roots.

The *objectid* field is an i-node number. Each object in a FS tree has many keys, but at minimum, *INODE\_ITEM* and *INODE\_REF* keys are present. *INODE\_ITEM* contains basic information about an inode: permissions, size, major and minor numbers, flags and count of the number of links to the object. *INODE\_ITEM* has always the lowest key of the object. The *offset* field of an i-node is zero and the *type* field is of value one. The key is therefore:

$$(inode, INODE\_ITEM, 0)$$

The i-node item does not store embedded data or extended attribute data as those are stored in separate items. The *INODE\_REF*'s key is:

$$(inode, INODE\_REF, parents\_dir\_inode)$$

This key stores a name of the object and its index number. The name of the object is the name of the file or directory and the index number is an index within the directory the object belongs to (see below).

The *EXTENT\_DATA* key is used for tracking all extents that a file's data belong to. The key is:

$$(inode, EXTENT\_DATA, logical\_offset)$$

The *logical\_offset* field is an offset into a file where data from the extent starts. Sequence of these items forms data of the file. Also, the item holds a logical offset into the extent on disk and a number of blocks in the extent. This is for being able to write in the middle of the extent. An optimization for small files (those that are smaller than one leaf block) is present. Small files (their sizes are less than size of the leaf node) are stored directly in a leaf node in order to access them efficiently.

There are two items that forms content of directories: *DIR\_ITEM* and *DIR\_INDEX*. The former's key is:

$$(inode, DIR\_ITEM, name\_hash)$$

There is a hash of the object's name in the offset field. It is used for fast lookup in a directory by a name. The latter item's key is:

$$(inode, DIR\_INDEX, index)$$

The offset field's content is an index of the object withing the directory (typically indexed by creation time of the object). It is used for bulk operations on directory like copying because indexes approximate data order on disk and thus saves disk seek times. Both items point to the same key. The referenced key is either *INODE\_ITEM* in case of file or directory, or *ROOT\_ITEM* in case of subvolume. Figure 7.1 depicts structures of extent and FS trees<sup>2</sup>.

### 4.2.3 Log Tree

The log tree is used for storing modified data and metadata of particular file on *fsync* operation. *fsync* is used to flush modified data of particular file to disk. In case of a system failure, the log is read to recovery potentially lost data.

---

<sup>2</sup>The picture is inspired by [https://btrfs.wiki.kernel.org/index.php/Data\\_Structures](https://btrfs.wiki.kernel.org/index.php/Data_Structures)

## 4.3 Using the Filesystem

This section [16] provides a brief overview of how to control the Btrfs filesystem via userspace tools.

### 4.3.1 Creating the Initial Volume

To create a volume consisting of multiple devices with default options, run:

```
# mkfs.btrfs /dev/sda /dev/sdb /dev/sdc
```

The default options are that metadata are replicated among all devices (if only one device is used, data are replicated within that device), if one copy gets corrupted there is a chance that one of other copies is left intact. Data on the other hand are spread among all devices.

To create a volume consisting of multiple devices with metadata being spread among all devices and data being mirrored on all devices, use:

```
# mkfs.btrfs -m raid0 -d raid1 /dev/sda /dev/sdb /dev/sdc
```

If one does not want to have metadata replicated, *-m single* can be used. This is, however, dangerous and not recommended.

### 4.3.2 Checking the Initial Volume

To show what devices make up a volume, run (all commands below are equivalent):

```
# btrfs filesystem show /dev/sda
# btrfs filesystem show /dev/sdb
# btrfs filesystem show /dev/sdc
```

After mounting the volume, space usage, and other useful information like how the volume was created, can be retrieved by running (note that argument is the mounting point, not the device):

```
# btrfs filesystem df /mnt
```

### 4.3.3 Resizing the Volume

Resizing the volume is an online operation, there is no need to unmount the volume. To shrink a volume, run (note that there is a minimum size of a volume and the filesystem does not allow the user to go beyond that limit):

```
# btrfs filesystem resize -500m /mnt
```

The opposite operation is done by growing the volume like so (instead of +500m, *max* can be used to grow the volume up to maximum space available):

```
# btrfs filesystem resize +500m /mnt
```

### 4.3.4 Adding and Removing Devices

Adding and removing a device is done by running (again, those are online operations):

```
# btrfs device add /dev/sdd /mnt
# btrfs device remove /dev/sdd /mnt
```

Adding a device to the volume must be completed with preparing the new device (rebalance/mirror the metadata and the data among devices):

```
# btrfs filesystem balance /mnt
```

### 4.3.5 Subvolumes and Snapshots

Subvolumes are basically directories that can be mounted.

#### Creating and Removing a Subvolume

```
# btrfs subvolume create SV1
# btrfs subvolume delete SV1
```

#### Listing Subvolumes

```
# btrfs subvolume list /mnt
```

#### Setting a Default Subvolume

Default subvolume is the one with number 0. This can be changed using:

```
# btrfs subvolume set-default 261 /mnt
```

#### Creating a Snapshot

To create a snapshot, use:

```
# btrfs subvolume snapshot / /snap-2012-11-19
```

The content of the snapshot remains the same as it was in time when the snapshot was created. Snapshots use almost no disk space once they are created. Only when the snapshot (they are writable) or the original filesystem are changed, then the snapshot occupies additional disk space (COW). When one deletes a file in the original filesystem, the removed storage cannot be claimed as a free since it is still present in the snapshot. Removing and listing snapshots are managed via the same commands like subvolumes.

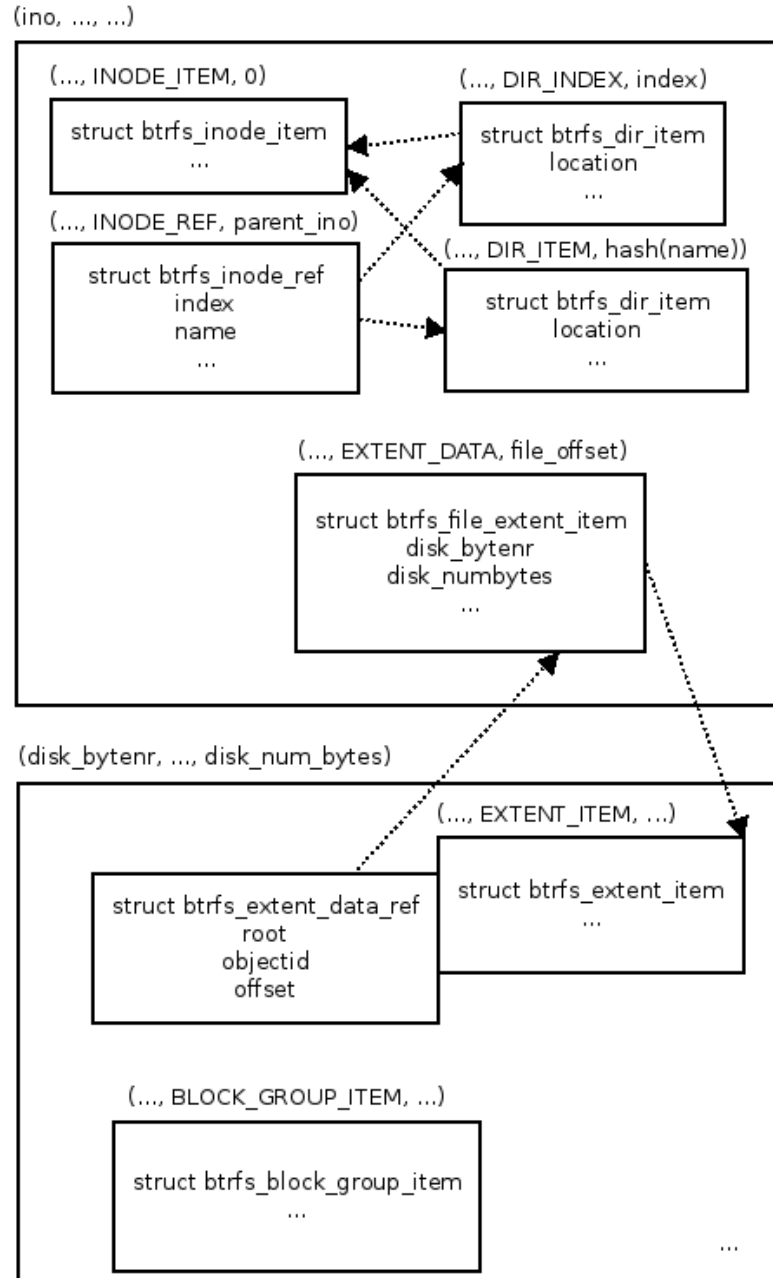


Figure 4.2: A simplification of extent and FS trees structures.

## Chapter 5

# Data Deduplication

This chapter focuses on the approaches dealing with data deduplication [17] [18], their advantages and disadvantages. Different types of deduplication according to different criteria is discussed. List of data deduplication use cases follows. ZFS filesystem has data deduplication implemented, an overview of how it's done in this filesystem is presented. Btrfs proof-of-concept offline deduplication feature description concludes the chapter.

### 5.1 Overview

Data deduplication is a process of locating duplicate sets (e.g. blocks) of data, leaving only one unique copy and removing the rest. Removed data are replaced with a link to the one unique copy. Shared pages in memory could be used as an analogy; after the process of deduplication, blocks are shared between files. One could think of data deduplication as a data compression technique as data amount is decreased after the process. A typical example would be an e-mail attachment of size 10 megabytes that is sent to one hundred users and so the attachment is stored on the mail server one hundred times. If the mail storage is deduplicated, only one copy of that attachment is left untouched while the rest is removed, saving 990 megabytes of storage. Other example that demonstrates advantage of data deduplication is reduced network usage. Suppose backup of a storage is sent to a remote backup server. If backed up data are deduplicated, data are reduced in size and so less network bandwidth is used. Data can be deduplicated directly in a filesystem, in an application (the application knows its specialized data best so they can be deduplicated efficiently; an e-mail server deduplicates e-mail messages for instance) or in network devices.

### 5.2 Types of Deduplication

Types of data deduplication can be distinguished according to *when* deduplication occurs:

- offline (or post-process)
- online (or immediate),

Another criterion for distinguishing types of deduplication is according to *what* is deduplicated:

- files,
- blocks,
- bytes

A detailed description of particular types follows.

### 5.2.1 Offline

Offline, or post-process, deduplication is run on demand. This eliminates an overhead of finding duplicates before each write operation. The process can be run at times when the storage is not accessed that much so the performance is not affected. With offline deduplication, duplicates are written out to disk and removed only when the process is actually run. Duplicates are present until the process is run and storage space is therefore wasted. Offline deduplication is suited for, including but not limited to, use cases where data duplication takes a long time to hit. In that case, deduplication is triggered only once duplicates are present in the filesystem. Note that *offline* does not mean that the storage is unmounted, it means that the process is not run before data are written to disk.

### 5.2.2 Online

Online, or immediate, deduplication is run before data are written out to disk. This does not waste storage space as duplicates are not stored on disk at all. Online deduplication, of course, brings overhead of having to calculate a hash of data to be written and comparing it with all the other hashes and finding potential duplicates on the disk *before* writeback. On the other hand, online deduplication eliminates unnecessary write operation of duplicate data; note that metadata still need to be written.

### 5.2.3 File

The file level deduplication operates on whole files, obviously. So even if there are two almost identical large files (virtual machine images for instance) that only differ in a few megabytes out of several gigabytes, they do not get deduplicated. File level deduplication could be efficient when used on storage that contains video or image files, for instance.

### 5.2.4 Block

The block level deduplication operates on blocks, solving the flaw of the file level deduplication. Thus, it is suitable for deduplicating storage containing files that differ in only small amount of data. Virtual machine images are a good example as they often contain the same operating system but differ in a few applications that are installed on it. However, the block level deduplication has a disadvantage of breaking sequential layout of files which surely have impact on disk access performance. Note that deduplicated block may differ from filesystem block. The size of deduplication block is crucial in the performance of the block level deduplication system.

### 5.2.5 Byte

The byte level deduplication is most costly as it must determine where duplicate ends and unique data starts. This type of deduplication is best to be left on the application level as application knows its data best.

## 5.3 Use Cases

Throughout this chapter, several use cases for data deduplication were mentioned, to summarize, the most obvious use cases are:

- virtual machine images,
- mail servers,
- backup servers,
- source code repositories

Obviously, as it can be seen from the examples above, the deduplication technique is most effective and useful in storage where there are many copies of the same file or part of a file stored.

## 5.4 Finding Duplicates

When deduplicating data a method for comparing data is needed. It is done by creating an unique identification of each file or block (depending on the level of deduplication). The unique identification can be a hash created by hash algorithms like MD5 or SHA-1. The problem with hash functions is that there is a possibility of two different inputs having the same output (hash). Although that some algorithms have very low probability of collision, it is always non-zero. Therefore, deduplicating process might remove data, thinking it is duplicate while data are different and just have the same hash, resulting in data loss/corruption. Although the possibility is very low, the risk is there. The solution to this is to run byte-by-byte comparison of two duplicates to ensure that data are indeed of the same content.

## 5.5 ZFS Deduplication

ZFS has built-in deduplication [19] support since 2009. ZFS provides online block-level deduplication. Usage of this feature in ZFS is shortly presented in this section.

Suppose there is a pool called *pool1*, when one wants to enable deduplication on that pool, this is what needs to be done:

```
# zfs set dedup=on pool1
```

This allows enabling deduplication only on the pools where a user expects data replication to be high. ZFS uses the *sha256* hash algorithm which is cryptographically strong. However, if a user does not want to take risks, deduplication can be enabled with:

```
# zfs set dedup=verify pool1
```

This will do byte-by-byte comparison when potential duplicate is found. Also, it is possible to choose the `fletcher4` hashing algorithm, instead of `sha256`. `Fletcher4` is much weaker but also faster. Of course, this makes sense only with the `verify` option enabled as possibility of hash collision is much higher. Setting `fletcher4` as the hash algorithm is done as follows:

```
# zfs set dedup=fletcher4 , verify pool1
```

## 5.6 Btrfs Offline Deduplication

There is the offline deduplication [20] feature for Btrfs, created by Josef Bacik in 2011. The feature is not however merged in upstream and is somewhat a proof-of-concept. In this section, the feature will be presented.

### 5.6.1 Userspace Part

The user space tool is given a directory that is desired to be deduplicated. The tool goes through all files, divided in 64 kilobytes blocks, in that directory and all subdirectories and builds a red-black tree [2] of hashes of those blocks. A node of the tree is defined as follows:

```
struct file_entry {
    ino_t ino;
    unsigned long pos;
    unsigned long len;
    unsigned long physical;
    char *hash;
    int entries;
    int fd;
    struct file_entry *next;
    struct rb_node n;
};
```

#### **ino**

- an inode number of the file

#### **pos**

- a logical offset into the file

#### **len**

- a length of the block

#### **physical**

- 

#### **hash**

- a hash of content the block (used as a key in the RB tree)

#### **entries**

- number of blocks with the same hash

**fd**

- a file descriptor of the opened file

**next**

- next entry with a duplicate block

**n**

- it is a node of the RB tree

Nodes with the *entries* field equals to zero are pruned, leaving the tree in a state of having only nodes with blocks that have duplicates on disk. The duplicates are checked by byte-by-byte comparison to ensure that they are indeed identical. Once this is done, those entries are sent via *ioctl* to the kernel to do the actual deduplication.

### 5.6.2 Kernel Part

The *ioctl* is given a logical offset, a length and a list of file descriptors with their logical offset. The *ioctl* will then split the extent of the given file descriptor and link it with an extent of each file descriptor from the list. Then those extents are freed. Potential duplicates are compared by byte-by-byte comparison again to make sure they were not changed on the way to the kernel. There are a few limitations in the current implementation:

1. File data spanned across more than one extent, in that case defragmentation would be called (not implemented). After defragmentation, the process would be triggered again. The reason for this is that the extent needs to be pointed at by each i-node that corresponds to file descriptor from the list.
2. Extents across subvolumes. Those might already be snapshots of each other.
3. Any data transformation. Data that are encrypted or compressed are different on disk than read in userspace. This is where online deduplication is more appropriate as duplicates might be looked up after the compression/encryption is done and before data are written out to disk.

The implementation uses its own function to link extents. However, there is an *ioctl* already for that called *btrfs\_ioctl\_clone* which can be used for online deduplication as well.

# Chapter 6

## Analysis

In this chapter, the design of the online deduplication feature for Btrfs will be presented. There are several key issues that need to be considered and will be discussed. This chapter will give a solution on how to implement the feature. The design of the feature presented in this chapter was discussed on the official mailing list of the Btrfs filesystem <sup>1</sup>.

### 6.1 High-Level Feature Overview

In this section, a high-level design overview will be described to have an idea what the following steps in designing the feature will be. The online deduplication process starts just before data are to be written out to disk. First, for each block, a hash is calculated. Next, the hashes are looked up in the deduplication table, that stores all hashes of on-disk blocks, as well as in a data structure that holds hashes of all in-memory blocks. If the hash is found, the duplicate is found and a reference is added to the existing block and the block will not be written to disk. If the deduplication system supports additional byte-by-byte comparison of blocks, blocks are compared once the hash is found in the deduplication table. Otherwise, the hash is not found and is added into the deduplication table and then the block is written out to the disk as usual.

### 6.2 Controls

There need to be several options to control the deduplication of the filesystem from the user stand-point, the options are as follows (means of controlling are mentioned in the brackets):

- to turn on/off deduplication (*mount* option)
- to specify a size of the deduplication block (*mount* option)
- to turn on byte-by-byte comparison of blocks that have same hashes (*mount* option)
- to fill the deduplication table with metadata; when deduplication is turned on a non-empty filesystem (*ioctl* call)

---

<sup>1</sup> <http://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg20669.html>

## 6.3 Limitations

There are several cases, albeit possible to implement, where deduplication is not triggered. However, some of them might be worth looking into in the future:

- *encryption* – Which is not implemented in Btrfs, at the time of writing the thesis.
- *inline/prealloc data* – Inline extents (less than filesystem block, see chapter 4) might not be worth the effort as storage space saving does not seem to be beneficial. Since inline extents are stored in-tree to gain performance when accessing it, it might be counter-productive to remove it from the tree and put it in a *file\_extent* so duplicates could point at it.
- *compression* – Deduplication is a kind of compression. While the two might work well together, one must decide which is run first, whether deduplication is run after compression or the other way around. If deduplication is run before compression, the cpu-heavy compression work could be saved as duplicates are not compressed. If deduplication is run after compression, the duplicates might be compressed into less blocks (or possibly single block) and less metadata would be written and less data would be compared with each other (finding duplicates). Either way, compression integration work is left for future improvements. Either compression or deduplication could be turned on.
- *NOCOW* – Deduplication, as well as with compression, would be exclusive with the NOCOW mode of the Btrfs filesystem.

## 6.4 Level of Deduplication

The first thing that needs to be decided is a level of deduplication. In the previous chapter 5, three levels of deduplication were presented. The most general approach, that is the block-level deduplication, seems to be suitable the most since it is the most general one. It offers a good compromise among the three. It is not that costly in terms of implementation as the byte-level and offers more granularity than the file-level deduplication. The pros and cons of the block-level deduplication were presented in the previous chapter.

## 6.5 Initiation of Deduplication

Two ways when to trigger a search for duplicates and/or the deduplication table update come to mind:

1. directly in the *write* system call
2. on the *sync* operation

The first option offers an easy implementation but will create a delay in performing the *write* in the userspace which is not desirable. Thus, it might be more appropriate to delay the process just before data are written out to disk to give data the chance to be changed. The delayed allocation time seems like a good candidate for initiation of deduplication. Data are not yet on-disk and the process might operate on them; create a hash, look it up in the table and according to the look up decide whether to write the block on disk or not.

## 6.6 Deduplication Block

According to [21], the practical values of deduplication block range from 8 to 64 kilobytes. The deduplication block will be 8192 bytes (two *btrfs* filesystem blocks). This would be default, however, a user would be able to set the block via *mount* option. Setting the deduplication block could be done on each *mount* differently, however, the user then lose the ability to deduplicate blocks of different block size that were hashed on one of the previous *mounts*.

Choosing the deduplication block is a very crucial decision since the block will have an impact on performance of the deduplication system. Having a small deduplication block increases the chance of having high deduplication ratio but will result in a need to write more metadata (*file\_extent\_item*'s that point to an extent or part of it).

So, for example, assume there is a 4 gigabyte file on disk its blocks are unique. Another file is about to be written differs only in several blocks from the previous one. Assume the deduplication block is set to 4096 bytes. There are 1048576 (4 gigabyte large file divided by the block size of 4096 bytes) hash items stored in the deduplication table that belong to the original file. Writing the file that differs only in a few blocks means creating the same number of *file\_extent\_item*'s (minus unique number of blocks) to reference the original blocks, which will decrease performance surely.

Therefore, the decision on the size of the deduplication block will depend on what data is stored in the storage.

There is a slight drawback to choosing the deduplication block different to the filesystem block. Assume two files, both are identical except that one of them contains additional 4 kilobytes of data in the beginning of the file. With the deduplication file of size 4 kilobytes, a file that will be written later will be successfully deduplicated. However, when deduplication is set to, e.g., 8 kilobytes, the first hashed block will be different in both files (because of the additional unique 4-kilobyte block in one of the files), as blocks of files are hashed from the beginning of the file. Therefore, blocks are hashed with an offset and so no duplicates are found even if the two files are almost identical.

## 6.7 Deduplication Table

The deduplication table is a data structure that holds hashes of all deduplication blocks. The key requirement here is to be able to look up a hash (find a duplicate) in the table as fast as possible. There are several possibilities of implementing the table in the *Btrfs* filesystem. The first obvious choice would be to use the checksum tree that holds block checksums of each extent. The problem with the checksum tree is that the checksums are looked up by a logical address for the start of the extent data. This is undesirable since it needs to be done the other way around. Logical addresses need to be looked up by a block hash. To solve this, a new tree will be created that will hold items that would have a hash (or part of it) as item's right-hand key value. This way, items could be looked up on a hash. The item of such key would contain information on where the block is stored. When a duplicate is found, that information would be used to track the block and increment the reference count. The item key could look as follows:

$$(\text{hash2}, \text{BTRFS\_DEDUP\_ITEM\_KEY}, \text{hash3})$$

The item's content would then consist of the following fields: *disk\_bytenr*, *disk\_num\_bytes*, *extent\_offset* and *len*. Fields *disk\_bytenr* and *disk\_num\_bytes* would point to the extent

item of the extent allocation tree. This would allow a reference to be incremented. The *extent\_offset* is the offset of the hashed block in the extent. The *len* field contains length of a hashed block, this field must be checked on a hash lookup as the deduplication block in the time of storing the hash might differ from the size at the time of the lookup. Although not listed above, one structure record must be devoted to store the remainder of the hash value (the first two parts are stored in the key). Looking up an item would be on the part in the key, then on hit, the remainder must be compared as well after putting the hash together from the key and the item.

### 6.7.1 Create

Creation of the deduplication table will consist of going through the filesystem and building the table by calculating hashes of each block. The table needs to be recreated when deduplication is turned on, off and on again. Also, when the deduplication block size is changed, the table needs to be recreated as well. Obviously, hashes of blocks of different size could not be used for duplicates lookup. Facilities already implemented for the compression feature (the kernel worker threads) might be of help here. Also, the scrub feature (scanning the filesystem for corrupted blocks and rewriting them with good copies, if available) could be considered as well as this is a operation heavy on CPU and should be done in the background.

### 6.7.2 Update

Updating the deduplication table will consist of two operations. A hash insertion will occur each time an unique block is written out to disk. Each time an extent is freed, hashes of all blocks of the extent will be removed from the deduplication table. These will ensure that the table is always up to date.

### 6.7.3 Performance

The performance of the deduplication table look up will be mainly affected by its location; whether it fits into the memory or not. Therefore, the more storage used, the more memory will be required for fast look up.

## 6.8 Hash Algorithm

*sha256* was chosen as the hash algorithm for its cryptographical strength; no collisions were found so far, even if there were collisions, additional byte-by-byte blocks comparison would cover that.

## 6.9 Referencing a Duplicate

Once a duplicate is found, a reference to original data is added. Finding a hash in the deduplication table will return a pointer to the *extent\_item* in the extent allocation tree. In the filesystem tree, a new item of type *EXTENT\_DATA* will be created with a key:

*(inode, EXTENT\_DATA, file\_offset).*

The *inode* field is an i-node number of the file that contains the duplicate that is meant to be written but is going to be thrown away instead and the *file\_offset* field is a position of the duplicate's block within that file. The item would then point to the same *extent\_item* in the extent allocation tree as the one from the *hash\_item* (the original on-disk block). The extent to be written is then split, leaving out the duplicate block. Increasing the reference count in the *extent\_item* is done via adding a *btrfs\_extent\_data\_ref* to it. The important thing is that previously mentioned *EXTENT\_DATA* points into the *extent\_item* only to a portion of the extent that the file actually contains. There might be a performance loss when there are too many references to the same extent. When implemented, this might be tested and if there is, indeed, a significant performance loss, number of references might be limited; therefore to write out a duplicate.

# Chapter 7

## Implementation

Challenges that were faced in the process of implementation of the online deduplication feature as well as data structures used will be presented in this chapter.

### 7.1 Initiating Deduplication

The deduplication is triggered at delayed allocation time, giving the data the chance to be changed. The function *run\_delalloc\_range* of *inode.c* allocates extent(s) for given range in the file. Within that function, there are several methods how the allocation is performed. Those are: NOCOW (Copy-On-Write is turned off), COW and compression. Deduplication fills in. NOCOW, deduplication (which uses COW) and compression are exclusive.

### 7.2 In-Memory Hashes

Obviously, hashes of in-memory blocks need to be stored as well as those on disk. Otherwise duplicates might (and will) be missed in the case when identical blocks are written within one sync/writeback operation. To solve this issue, a red-black tree is created within the *btrfs\_fs\_info* structure that holds hashes of in-memory blocks. Red-black trees provides logarithmic search, insert and delete operations and thus are fast enough for handling block hashes. Once a block is written out to disk, its hash is erased from the tree. For completeness, a structure that each node of the RB tree contains is shown:

```
struct btrfs_rb_dedup_hash {
    struct inode *inode
    u64 *hash;
    u64 file_offset;
    struct rb_node node;
};
```

#### **inode**

- an inode structure of the file containing the hashed block

#### **hash**

- a hash of the block

#### **file\_offset**

- an offset into the file where the block is located

#### **node**

- it is a node of the RB tree

The *inode* and *file\_offset* fields are used to read the block from the memory for the byte-by-byte block comparison.

### **7.3 Deduplication Table**

Looking up an item in the deduplication table that contains a location of the block is on the following key:

*(hash2, BTRFS\_DEDUP\_ITEM\_KEY, hash3)*

The key contains only a half of the hash value. The reason is that the hash value is 256 bits in size and both left-hand and right-hand key values are only 64 bits. In order to make sure the right item was found, an additional comparison needs to be done with the rest of the hash value that is stored in the item. The item data would contain the following structure:

```
struct btrfs_dedup_item {
    __le64 disk_bytenr;
    __le64 disk_num_bytes;
    __le64 extent_offset;
    __le64 len;
    __le64 hash0;
    __le64 hash1;
} __attribute__((packed));
```

#### **disk\_bytenr**

- an on-disk location of the extent containing the block

#### **disk\_num\_bytes**

- a length of the extent (*disk\_bytenr* and *disk\_num\_bytes* are the key into the extent allocation tree)

#### **extent\_offset**

- an offset into the extent containing the block

#### **len**

- length of the hashed block; if a duplicate is found, this is used for a check whether the block is of the same size (whether the deduplication block is set to the size of the duplicate)

#### **hash0 and hash1**

- a half of the hash value

The structure described above is sufficient for storing the deduplication metadata. However, in order to be able to remove an item from the table, another item and a key that the item is looked up on is introduced:

*(disk\_bytenr, BTRFS\_DEDUP\_REF\_KEY, extent\_offset).*

This will ensure that on an extent deletion, a hash of the block in the extent could be looked up on an on-disk location of the extent and an offset in the extent. The found hash value is then used to search for the item and remove it. The item is defined as follows:

```
struct btrfs_dedup_hash_item {
    __le64 hash0;
    __le64 hash1;
    __le64 hash2;
    __le64 hash3;
} __attribute__((packed));
```

Figure 7.1 depicts structures of the dedup tree and its relationship with the extent tree.

## 7.4 Writing Deduplication Metadata

In general, metadata should be written only after it is known that data are written on disk. Therefore, the filesystem does not end up with metadata pointing to non-existing data, or possibly data that should not be accessed by the file that belongs to the metadata.

The deduplication metadata consists of *struct btrfs\_dedup\_item* (see 6) that holds a block hash and a block location on disk. A *struct btrfs\_file\_extent\_item* that references an extent, or part of it, is considered as a deduplication metadata as well. The content of items of these structures is known at the delayed allocation time (not including on-disk location of in-memory blocks). However, as it was mentioned earlier in this section, metadata cannot be written yet. To solve this, the hashes of each block of the extent are added to the *btrfs\_ordered\_extent:ordered-data.h* structure. Later, in the *btrfs\_finish\_ordered\_io:inode.c* function, where metadata can be written because data are already on disk, hashes are inserted into the deduplication table (tree) as well. The actual structure holding a block hash follows:

```
struct btrfs_dedup_hash {
    u64 extent_offset;
    u64 *hash;
    struct list_head list;
};
```

### **extent\_offset**

- an offset into the extent where the block that will be written to disk is located

### **hash**

- a hash value of the block that is going to be written to disk

### **list**

- this means that this structure is an item in a linked list [2]

The duplicate metadata (data for referencing a duplicate) cannot be passed into the *btrfs\_ordered\_extent* structure. The reason is that it might happen that the whole extent consist only of duplicate blocks and so no such structure is created. For that reason, the duplicate metadata are inserted into a linked list inside the *btrfs\_fs\_info:ctree.h* structure. The metadata are then written once all in-memory blocks that should have been referenced are on disk as well, so they actually can be referenced. The duplicate metadata structure follows:

```
struct btrfs_duplicate {
    struct inode *inode;
    u64 disk_bytenr;
    u64 num_bytes;
    u64 extent_offset;
    u64 file_offset;
    u64 *inram;
    struct list_head list;
};
```

#### **inode**

- an inode structure of the file containing a duplicate that will be thrown away and will point to the existing block instead

#### **disk\_bytenr**

- a location on disk of the extent containing the block that will be referenced (left-hand key value of an item of the extent tree)

#### **num\_bytes**

- size of the extent containing the referenced block (right-hand key value of an item of the extent tree)

#### **extent\_offset**

- an offset into the extent containing the referenced block

#### **file\_offset**

- an offset into a file containing the duplicate that will be thrown away

#### **inram**

- this item contains a hash of a block that will be referenced
- it is set only in the case that the block to be referenced is not on disk
- if this item is set, items *disk\_bytenr*, *num\_bytes*, *extent\_offset* are not set because those values are not known yet and will be retrieved and set once known; at finishing IO time when metadata are written

#### **list**

- this means that this structure is an item in a linked list [2]

## 7.5 Actual Deduplication

The actual deduplication occurs in the function `cow_file_range_dedup:inode.c` using facilities presented earlier in this chapter.

## 7.6 Controlling Deduplication

### 7.6.1 Turning Deduplication On

Deduplication is turned on by a `mount` option as follows:

```
mount -o dedup device
```

Once deduplication is turned on, off and on again, the blocks written in the period when deduplication was turned off won't be stored in the deduplication table and cannot be identified as duplicates (see 7.6.4 for similar topic).

### 7.6.2 Setting Deduplication Block Size

```
mount -o dedup -o dedupsize=16384 device
```

The value of the `dedupsize` option is in bytes. By default, the deduplication block is set to 8192 bytes. Setting a different block size than the one that had been set already will result in all records in the deduplication table being unusable since those are hashes for blocks of different block size. (Note that `-o dedup` must be used as well, otherwise `-o dedupsize` will have no effect.)

### 7.6.3 Turning byte-by-byte Block Comparison On

```
mount -o dedup -o dedupverify=on device
```

By default, byte-by-byte block comparison is turned off; `dedupverify` is set to `off`. (Note that `-o dedup` must be used as well, otherwise `-o dedupverify` will have no effect.)

### 7.6.4 Syncing Deduplication Metadata

There is an `ioctl` system call, `BTRFS_IOC_DEDUP_SYNC`. This `ioctl` goes through the filesystem and fills the dedup tree with metadata. It reads each file by blocks of size of the deduplication block, hash them and store the hashes in the dedup tree. The `ioctl` will use the deduplication block size specified by the `mount` option. This is useful once users desire to turn on deduplication on non-empty filesystems. Note that the metadata sync cannot be used (as stated in 7.6.1) when deduplication is turn on, off and on again because the sync does not remove hashes of removed blocks. However, that can be fixed, there are two options how to do that:

1. before sync, clean all data from the dedup tree
2. go through the dedup tree and check whether data from `dedup_item` are still valid; that would require locating data, calculating its hash and compare the hash with the hash from the `dedup_item`

Option number 1 seems to be a better choice as it does not require additional work of going through the dedup tree.

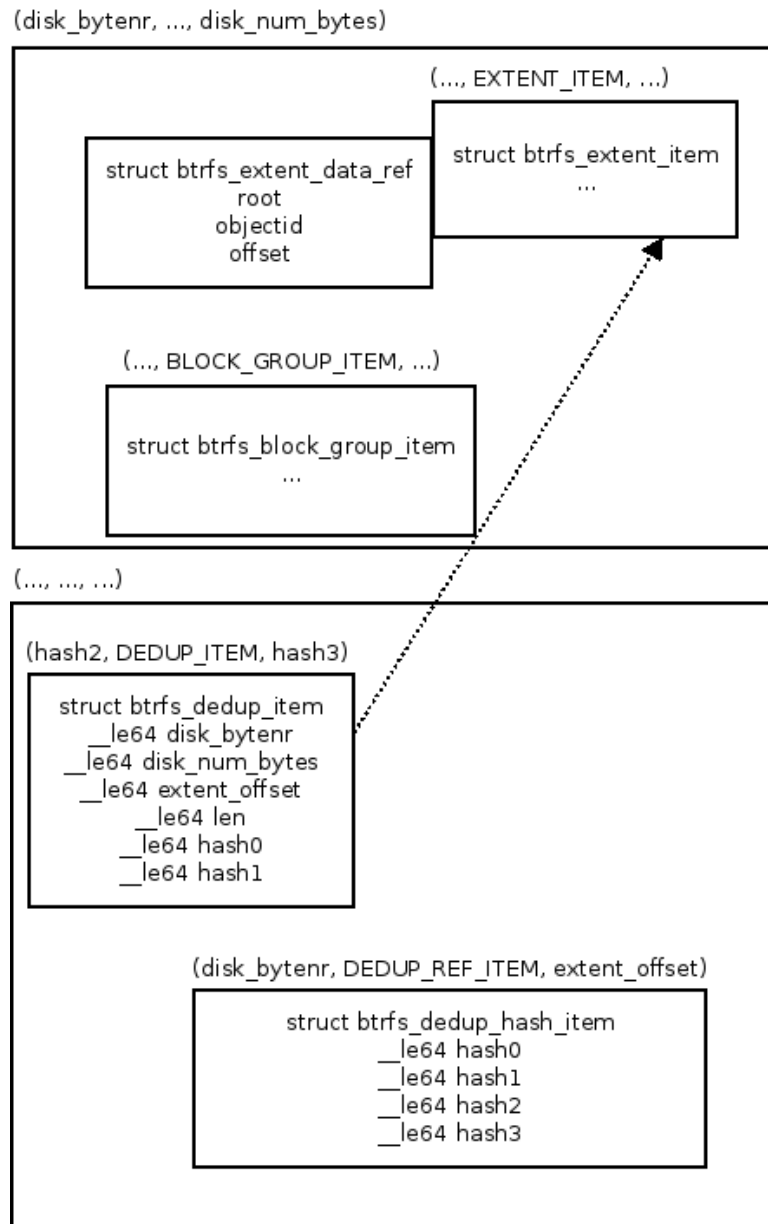


Figure 7.1: A simplification of the dedup tree and its relationship with the extent tree.

# Chapter 8

## Recent work

In the time of working on the thesis, one month before handing in the thesis, a patch for online deduplication was sent <sup>1</sup> on the Btrfs mailing list. Despite the fact that the author of the thesis sent his design proposal on the very same mailing list before finishing the term project, the author of the patch did not contacted him and worked on the solution himself. Therefore, the solutions are similar. In this section, the patch will be presented and comparison between the solution of this thesis and the patch sent on the mailing list will be given. Note: this section describes the second version <sup>2</sup> of the patch as it is the latest version in the time of writing the thesis.

### 8.1 Implementation

There is a dedicated tree for storing deduplication metadata, two new keys are introduced. The first key, *BTRFS\_DEDUP\_ITEM\_KEY*, is of form:

$$(hash3, BTRFS_DEDUP_ITEM_KEY, disk.bytenr)$$

The *hash3* field is a 64 bit long part of the block hash value, *disk.bytenr* is an on-disk location of the hashed block. The item's structure is defined as follows:

```
struct btrfs_dedup_item {
    __le64 len;
} __attribute__((__packed__));
```

The only field in the structure is length of the hashed block. The rest of the hash value is stored in the item's data in the leaf. The item is used for looking up duplicates. Duplicates are looked up on the block's content hash value. The *disk.bytenr* part of the key is not used for a look up, it is used as a return value of on-disk location of the block instead. The second key, *BTRFS\_DEDUP\_INDEX\_KEY*, available in the tree is:

$$(BTRFS_DEDUP_OBJECTID, BTRFS_DEDUP_INDEX_KEY, disk.bytenr)$$

The item's structure is defined as follows:

```
struct btrfs_dedup_hash_item {
    __le64 hash[BTRFS_DEDUP_HASHLEN];
} __attribute__((__packed__));
```

---

<sup>1</sup><http://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg23656.html>

<sup>2</sup><http://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg23809.html>

The item's only field is a block's content hash value. This item is used on freeing an extent, the item's are looked up on their on-disk location. The *objectid* field of an item remains the same for all items.

## 8.2 Features

The implementation sent to the Btrfs mailing list is block-level kind of deduplication. It has a support for variable deduplication block size. One can set the deduplication block size via mount option e.g. as follows:

```
mount -o dedup, dedup_blocksize=X /dev/sda /mnt
```

The default deduplication block size is 8192 (bytes). As shown above, deduplication is turned on with the *dedup* mount option. There is a second step to enable deduplication by executing:

```
$ btrfs filesystem sync /mnt
```

The *sha256* hash algorithm is used for creating hashes of blocks. The implementation uses the kernel work threads for cpu-heavy deduplication work. In the time of writing the thesis, byte-by-byte block comparison was a planned feature as well as an *ioctl* call for userspace application (*btrfs-progs*, in most linux distributions).

## 8.3 Comparison

This section compares features of both implementations that has been implemented as of the time of writing the thesis.

While the two implementations are very similar, possibly based on the same design proposal, there are some differences and both have their advantages and disadvantages. The mailing list implementation mainly benefits (performance-wise) from its use of kernel worker threads for delayed allocation work and thus for heavy on cpu deduplication work. From the implementation stand-point, the mailing list implementation has its own *cow\_file\_range* implementation making the design clearer and easier to read and the implementation is possibly faster; the thesis implementation creates a wrapper on top of the original *cow\_file\_range* function. While the thesis implementation lacks of benefits from running deduplication in kernel threads, it adds the byte-by-byte block comparison feature. The feature has an impact on data integrity. The thesis implementation also benefits from ability to deduplicate in-ram blocks.

## 8.4 Summary

This section summarizes comparison made in the previous section.

	<b>The mailing list impl.</b>	<b>The thesis impl.</b>
<b>Data structure for dedup. table</b>	B-tree	B-tree
<b>Byte-by-Byte comparison</b>	No	Yes
<b>Threads</b>	Yes	No
<b>Userspace support</b>	No	No
<b>Variable block size</b>	Yes	Yes
<b>Hash algorithm</b>	sha256	sha256
<b>Filesystem scan</b>	No	Yes
<b>Compression integration</b>	No	No
<b>Multiple hash algorithm support</b>	No	No
<b>Turn on/off dedup. on subvolumes</b>	No	No

# Chapter 9

## Testing and Results

In this chapter, an overview of possible test cases to test the online deduplication feature will be presented. Some of them were used to actual testing and those will be presented along with results. A short description of how the tests were performed and environment used for development will conclude the chapter.

### 9.1 Test Case 1: Disk Usage

#### 9.1.1 Overview

In this test case, a whole-file deduplicate is written to perform a simple check that deduplication works and that *btrfs fi df* shows the correct value.

#### 9.1.2 Test

```
# mkfs.btrfs /dev/sda
# mount -o dedup /dev/sda /mnt/
$ btrfs fi df /mnt/
  Data: total=8.00MB, used=256.00KB
  System, DUP: total=8.00MB, used=4.00KB
  System: total=4.00MB, used=0.00
  Metadata, DUP: total=1.00GB, used=28.00KB
  Metadata: total=8.00MB, used=0.00
$ dd if=/dev/zero of=/mnt/file bs=4K count=1; sync
$ dd if=/dev/zero of=/mnt/file bs=1M count=10; sync
$ btrfs fi df /mnt/
  Data: total=1.01GB, used=260.00KB
  System, DUP: total=8.00MB, used=4.00KB
  System: total=4.00MB, used=0.00
  Metadata, DUP: total=1.00GB, used=432.00KB
  Metadata: total=8.00MB, used=0.00
```

#### 9.1.3 Discussion

A file of size one filesystem block, 4 kilobytes, is created. Then, another file of size 10 megabytes is created. The content of the latter file consists of blocks identical to the block

of the former file, making it a duplicate. Therefore, only 4 kilobytes of data should be written to disk. Output of the command `btfs fs df mount-point` confirms that.

## 9.2 Test case 2: Read – Metadata Fragmentation

While the read operation is not affected by the deduplication process itself, it is affected in certain situations by the state of metadata that the deduplication system creates as was described in 6.6. In this test case, there is a file written on disk twice. Deduplication is turned on, so the second copy are just links to the blocks of the first copy. The speed of accessing both files is measured.

## 9.3 Test case 3: Read – Data Fragmentation

In this test case, there are several non-duplicate, unique, files on disk. Another file is created, a duplicate consisting of first blocks of each unique file mentioned earlier making it a file with data stored non-sequentially. The speed of accessing the duplicate with fragmented data and the file of the same content stored sequentially (deduplication is turned off) is measured.

## 9.4 Test case 4: Write Unique Block

In the terms of writing data while deduplication is on, there are two situations that are obvious to test. The first one is to write a file all of its blocks are duplicates. The second test case is to do the exact opposite, to write a file all of its blocks are unique. Those are the corner cases and thus it makes sense to test them. The latter test case is explored in this section. When compared to writing a file with deduplication off, the process is slowed down by calculating hashes of the blocks and looking up duplicates in the tree. Another possible performance bottleneck can be byte-by-byte block comparison in the case that a false duplicate is found (a key part of the hash is of the same value with another block).

## 9.5 Test case 5: Write Duplicate Block

This is a follow-up to the previous test case (9.4), writing a duplicate file is the subject to test here. Because no blocks will be actually written, the process of „writing“ the file should be theoretically faster than writing an unique file. Nevertheless, looking for duplicates in the deduplication table will, on the other hand, slow the process down. If the byte-by-byte comparison is turned on, the process will be even slower. All these cases will be explored in the following subsections.

## 9.6 Testing environment

Tests were performed in a virtual machine environment. For virtualization, *KVM* was used. The host and guest were of the configuration listed in following tables:

<b>Architecture</b>	x86_64
<b>CPU</b>	Intel Core 2 Duo 2.0 GHz
<b>Memory</b>	2 GB
<b>HDD</b>	120 GB, 5400 rpms
<b>OS</b>	Fedora 17

Table 9.1: The host machine configuration

<b>Architecture</b>	x86_64
<b>Memory</b>	768 MB
<b>OS</b>	Fedora 18
<b>Kernel</b>	3.9

Table 9.2: The guest machine configuration

## 9.7 Testing

The write performance was measured on a 10-megabyte random-content file created like so:

```
dd if=/dev/urandom of=test10 bs=10M count=1
```

The deduplication process occurs on the *sync* operation so that operation is measured with the *time* command. First, the file is copied on a btrfs partition with deduplication turned on. Then, the same file is copied there again; a duplicate is created. Measuring writing unique and duplicate files is performed as follows (*sync* is run even before copying to lower the possibility of other data being synced):

```
sync; cp test10 mnt/; time sync
sync; cp test10 mnt/file; time sync
```

## 9.8 Development Testing

During development, simple test cases were performed to verify that the implementation works as expected. Those test cases included writing small files (a few blocks) where various combination of their blocks were duplicates. The test cases were not automated. Instead, *btrfs-debug-tree* (that is part of a userspace suite *btrfs-progs*) was used to print information about written extents to see whether the blocks were written (or not) as expected.

## 9.9 Summary

Testing showed that, as predicted in 6, choosing the deduplication blocks is crucial performance-wise. Deduplication block size of one filesystem block (4 kilobytes) does bring poor results when writing a file consisting of duplicates blocks only due to metadata fragmentation. From table 9.3 it can be seen that performance gain is achieved when the deduplication block is set to 32 and 64 kilobytes. In those cases, writing a duplicate file is faster on *sync* than writing a unique file.

Although not listed in the table, testing a 100-megabyte file (same test cases) with the deduplication block set to 64 kilobytes did not bring the same results; writing both unique and duplicate files resulted in approximately same time. This finding suggests that the deduplication system might be well suited for storage with rather small files like source code repositories, backup servers (depends on what we back-up, obviously) or mail servers.

Test results are summarized in table 9.3. Note that although test cases were performed several times they are not very accurate as the *sync* operation might have handled other files than the test ones as well. Nevertheless, the results shows approximate performance loss/gain in the deduplication system.

<b>Test case/Dedup block size [kB]</b>	<b>4</b>	<b>8</b>	<b>16</b>	<b>32</b>	<b>64</b>
Write Unique Block	0.9s	0.8s	0.8s	0.8s	0.7s
Write Unique Block (dedupverify=on)	1.2s	1.0s	1.0s	1.0s	0.8s
Write Duplicate Block	14.1s	13.7s	1.9s	0.5s	0.3s
Write Duplicate Block (dedupverify=on)	20.0s	14.2s	3.1s	1.0s	1.0s

Table 9.3: Test results summary

# Chapter 10

## Future Work

The current state of the online deduplication feature for Btrfs leaves several opportunities for improvement in the future. This chapter will summarize them and try to point out potential problems and implementation suggestions.

Basically, deduplication is a kind of compression as it saves storage space. However, it makes sense to integrate deduplication and Btrfs's transparent compression (LZO, zlib). As it was pointed out earlier in the thesis (see 6.3), it needs to be decided whether to trigger deduplication before or after the compression process. Both approaches have their advantages and disadvantages that were presented in 6.3.

The compression code already provides kernel work threads that do compression on *sync* or general writeback. These kernel threads could be used very well for online deduplication as they are suited for CPU heavy operations. A framework for submitting data (*struct bio* [2]) would have to be build, as it is done the same way in the compression code.

Currently, the deduplication process is triggered on the root subvolume (the one with *subvolid=0*), and therefore on the whole filesystem. However it makes sense for a user to turn on deduplication on particular subvolume(s) as he/she more less knows where deduplication occurs and so can set the feature only on those subvolumes not to decrease performance on others.

The *sha256* hashing algorithm is used for creating unique identifications of deduplication blocks. While the algorithm is cryptographically strong, it is not that fast. Thus, it might be useful to give a user a choice of hashing algorithm to speed the hashing up. Choosing a faster hashing algorithm and therefore probably with a higher probability of collision would need to triggered byte-by-byte block comparison by default not to put user's data to risk losing their integrity. *btrfs-progs*<sup>1</sup> is a userspace application that provides means of controlling btrfs filesystems. The online deduplication feature would benefit from its support in *btrfs-progs*. From the user stand-point, there are a few possibilities for improvement that follow:

- specify a hash algorithm
- specify subvolumes to turn on/off deduplication on
- print deduplication ratio

---

<sup>1</sup>[https://btrfs.wiki.kernel.org/index.php/Btrfs\\_source\\_repositories#btrfs-progs\\_git\\_repository](https://btrfs.wiki.kernel.org/index.php/Btrfs_source_repositories#btrfs-progs_git_repository)

Controlling options that are already implemented (either by *mount* or *ioctl*) can be added to the userspace tool as well, these include the following:

- specify deduplication block size
- scan a filesystem and create the deduplication table
- turn on/off byte-by-byte block comparison

# Chapter 11

## Conclusion

This thesis dealt with the online deduplication feature for Btrfs. The feature was successfully designed and implemented. Analysis of the online deduplication feature for Btrfs was the key part of the thesis as the author faced challenges when designing data structures that deduplication systems use. The main data structure is deduplication table for storing unique identifications of on-disk data blocks. The copy-on-write friendly b-tree was chosen as a data structure for the deduplication table making the lookup efficient. The design proposal was sent to the Btrfs mailing list, the author received useful feedback and the feature was re-designed where appropriate. (6)

The online deduplication feature was implemented for Btrfs. It is a block level kind of deduplication. It has support for variable deduplication block sizes; users can set the block size to suit their needs. Without this support, the feature would be barely usable. Online deduplication uses *sha256* for creating hashes; unique identification of blocks and it supports byte-by-byte block comparison to avoid hash collisions. (7)

There is probably no ultimate answer to whether the online deduplication for Btrfs is useful or not. Users need to know their data stored in the filesystem they want to turn on deduplication on. As testing showed, choosing the deduplication block size is very crucial in the deduplication performance. Generally speaking, the deduplication block should be chosen according to the file sizes. A filesystem consisting of small files (possibly duplicates, obviously) would benefit from the deduplication feature with the deduplication block set to a size similar to the filesystem block (4 kilobytes in Btrfs by default). Having a filesystems with files large in size and the deduplication block set to the filesystem block size could decrease performance significantly. Testing showed that the implemented deduplication system seems to be best suited for storage with rather smaller files like source code repositories, backup server (depends on what data are backed up, obviously) or mail servers. (9)

Despite the fact that the author of the thesis announced the project on the mailing list, another implementation of online deduplication was sent to the very same mailing list a month before handing in the thesis. The two implementations were compared in the thesis, both having their advantages and disadvantages. It was decided not to send the thesis implementation to the mailing list. (8)

While the planned features were implemented, there is still a work to do to improve the implementation. From the user-stand point, there are several options how to extend the userspace application for Btrfs (*btrfs-progs*). The minimal of set of improvements would include turning on deduplication, setting a deduplication block size or setting byte-by-byte block verification; the controls now available only through the *mount* command. From

the kernel side, there are features like compression integration or multiple hash algorithm support, that can be added to the existing implementation. (10)

# Bibliography

- [1] *Ext4 How to* [online], accessed 2012-11-16.  
[https://ext4.wiki.kernel.org/index.php/Ext4\\_Howto](https://ext4.wiki.kernel.org/index.php/Ext4_Howto).
- [2] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, third edition, 2010.
- [3] Silicon Graphics International Corp. *XFS: A high-performance journaling filesystem* [online], 2012 [accessed 2012-11-16]. <http://oss.sgi.com/projects/xfs/>.
- [4] Silicon Graphics Inc. *XFS User Guide: A guide for XFS filesystem users and administrators*. [online], 2006 [accessed 2012-11-16]. [http://xfs.org/docs/xfsdocs-xml-dev/XFS\\_User\\_Guide//tmp/en-US/html/index.html](http://xfs.org/docs/xfsdocs-xml-dev/XFS_User_Guide//tmp/en-US/html/index.html).
- [5] *XFS* — *Wikipedia, The Free Encyclopedia* [online], accessed 2012-11-16.  
<http://en.wikipedia.org/wiki/XFS>.
- [6] *Reiser4* — *Wikipedia, The Free Encyclopedia* [online], accessed 2012-11-16.  
<http://en.wikipedia.org/wiki/Reiser4>.
- [7] *ReiserFS* — *Wikipedia, The Free Encyclopedia* [online], accessed 2012-11-16.  
<http://en.wikipedia.org/wiki/ReiserFS>.
- [8] *What is ZFS?* [online], accessed 2012-11-16.  
<http://hub.opensolaris.org/bin/view/Community+Group+zfs/whatis>.
- [9] *Comparison of filesystems* — *Wikipedia, The Free Encyclopedia* [online], accessed 2013-01-03. [http://en.wikipedia.org/wiki/Comparison\\_of\\_file\\_systems](http://en.wikipedia.org/wiki/Comparison_of_file_systems).
- [10] Ohad Rodeh. *B-trees, Shadowing, and Clones*. *ACM Transactions on Computational Logic*, 2007.
- [11] *Btrfs developer documentation* [online], accessed 2012-07-07. [https://btrfs.wiki.kernel.org/index.php/Main\\_Page#Developer\\_documentation](https://btrfs.wiki.kernel.org/index.php/Main_Page#Developer_documentation).
- [12] Josef Bacik. *Btrfs: The Swiss Army Knife of Storage*. *USENIX ;login.*, 2012.
- [13] *Btrfs* — *Wikipedia, The Free Encyclopedia* [online], accessed 2012-11-16.  
<http://en.wikipedia.org/wiki/Btrfs>.
- [14] Chris Mason Odah Rodeh, Josef Bacik. *Btrfs: The Linux B-Tree Filesystem*. *IBM Research Report RJ10501*, 2012.
- [15] *Btrfs Trees* [online], accessed 2013-04-24.  
<https://btrfs.wiki.kernel.org/index.php/Trees>.

- [16] *Btrfs Fun – Funtoo Linux Wiki* [online], accessed 2012-07-08.  
[http://www.funtoo.org/wiki/BTRFS\\_Fun](http://www.funtoo.org/wiki/BTRFS_Fun).
- [17] *Data deduplication* — *Wikipedia, The Free Encyclopedia* [online], accessed 2012-11-18. [http://en.wikipedia.org/wiki/Data\\_deduplication](http://en.wikipedia.org/wiki/Data_deduplication).
- [18] Jeff Hawkins Stephen J. Bigelow. *Data deduplication definition* [online], accessed 2012-11-16.  
<http://searchstorage.techtarget.com/definition/data-deduplication>.
- [19] Jeff Bonwick. *ZFS Deduplication* [online], 2009 [accessed 2012-11-17].  
[https://blogs.oracle.com/bonwick/entry/zfs\\_dedup](https://blogs.oracle.com/bonwick/entry/zfs_dedup).
- [20] Josef Bacik. *Offline Deduplication for Btrfs* [online], 2011 [accessed 2012-07-07].  
<http://lwn.net/Articles/422331/>.
- [21] Dutch T. Meyer and William J. Bolosky. *A Study of Practical Deduplication*. In *9th USENIX conference on File and storage technologies*, pages 1–1, February 2011.

# Appendix A

## CD content

### README

- contains installation instruction

### xkrize06\_dip.pdf

- the thesis in the PDF format

### src/

- source code of the *btrfs* module for the Linux kernel and *btrfs-progs*

### patches/

- patches for the Linux kernel and *btrfs-progs* that add online deduplication

### tex/

- L<sup>A</sup>T<sub>E</sub>X source code of the thesis

# Appendix B

## Manual

### B.1 Installation

Since there are many ways on how to configure the Linux kernel and there are many Linux distributions, there is no general way of instructions on how to build it. Build the Linux kernel according to your needs and what your distribution requires and apply the patches for online deduplication.

Building and installing *btrfs-progs* is done as follows:

```
$ make
```

```
# make install
```

## B.2 Usage

This section shows how to control deduplication using the *mount* command.

Create a *btrfs* partition.

```
# mkfs.btrfs -f /dev/sda
```

Turn on deduplication.

```
# mount -o dedup /dev/sda /mnt
```

Set deduplication block size to 8192 bytes.

```
# mount -o dedup,dedupsize=8192 /dev/sda /mnt
```

Turn on byte-by-byte block comparison.

```
# mount -o dedup,dedupverify=on /dev/sda /mnt
```

Set deduplication block size to 16384 bytes and turn on byte-by-byte block comparison.

```
# mount -o dedup,dedupsize=16384,dedupverify=on /dev/sda /mnt
```

When deduplication turned on a non-empty filesystem, one can run the following *ioctl*:

```
BTRFS_IOC_DEDUP_SYNC
```

(Note that the *ioctl* does not work when deduplication is turned on, off and on again, as described in the thesis.)