Btr-Diff: An Innovative Approach to Differentiate BtrFs Snapshots

Nafisa Mandliwala
Pune Institute of Computer Technology
nafisa.mandliwala@gmail.com

Swapnil Pimpale
PICT LUG
pimpale.swapnil@gmail.com

Narendra Pal Singh
Pune Institute of Computer Technology
narendrapal2020@gmail.com

Ganesh Phatangare
PICT LUG
gphatangare@gmail.com

Abstract

Efficient storage and fast retrieval of data has always been of utmost importance. The BtrFs file system is a copy-on-write (COW) based B-tree file system that has an built-in support for snapshots and is considered a potential replacement for the EXT4 file system. It is designed specifically to address the need to scale, manage and administer large storage configurations of Linux systems. Snapshots are useful to have local online "copies" of the file system that can be referred back to, or to implement a form of deduplication, or for taking a full backup of the file system. The ability to compare these snapshots becomes crucial for system administrators as well as end users.

The existing snapshot management tools perform directory based comparison on block level in user space. This approach is generic and is not suitable for B-tree based file systems that are designed to cater to large storage. Simply traversing the directory structure is slow and only gets slower as the file system grows. With the BtrFs send/receive mechanism, the filesystem can be instructed to calculate the set of changes made between the two snapshots and serialize them to a file.

Our objective is to leverage the `send` part in the kernel to implement a new mechanism to list all the files that have been added, removed, changed or had their metadata changed in some way. The proposed solution takes advantage of the BtrFs B-tree structure and its powerful snapshot capabilities to speed up the tree traversal and detect changes in snapshots based on inode values. In addition, our tool can also detect changes between a snapshot and an explicitly mentioned parent. This lends itself for daily incremental backups of the file system, and can very easily be integrated with existing snapshot management tools.

1 Introduction

In current times, where data is critical to every organization, its appropriate storage and management is what brings in value. Our approach aims at taking advantage of the BtrFs architectural features that are made available by design. This facilitates speedy tree traversal to detect changes in snapshots. The `send` ioctl runs the tree comparison algorithm in kernel space using the on-disk metadata format (rather than the abstract `stat` format exported to the user space), which includes the ability to recognize when entire sub-trees can be skipped for comparison. Since the whole comparison algorithm runs in kernel space, the algorithm is clearly superior over existing user space snapshot management tools such as Snapper[1].

Snapper uses `diff` algorithm with a few more optimizations to avoid comparing files that have not changed. This approach requires all of the metadata for the two trees being compared to be read. The most I/O intensive part is not comparing the files but generating the list of changed files. It needs to list all the files in the tree and `stat` them to see if they have changed between the snapshots. The performance of such an algorithm degrades drastically as changes to the file system grow. This is mainly caused because Snapper deploys an algorithm that is not specifically designed to run on COW based file systems.

The rest of the paper is organized as follows: Section 2 explains the BtrFs internal architecture, associated data structures, and features. Working of the send-receive code and the `diff` commands used, are discussed in

[1]The description takes into consideration, the older version of Snapper. The newer version, however, does use the send-receive code for `diff` generation

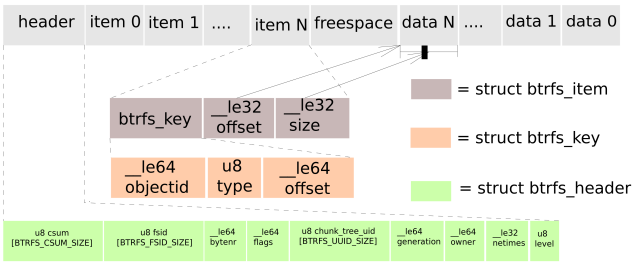

Figure 1: The Terminal Node Structure (src: BtrFs Design Wiki)

Section 3. Section 4 covers the design of the proposed solution, the algorithm used and its implementation. This is followed by benchmarking and its analysis in Section 5. Section 6 states applications of the send-receive code and `diff` generation. Section 7 lists the possible feature additions to Btr-diff. Finally, Section 8 summarizes the conclusions of the paper and is followed by references.

2 BtrFs File System

The Btrfs file system is scalable to a maximum file/file system size up to 16 exabytes. Although it exposes a plethora of features w.r.t. scalability, data integrity, data compression, SSD optimizations etc, this paper focuses only on the ones relevant to snapshot storage: comparison and retrieval.

The file system uses the 'copy-on-write' B-tree as its generic data structure. B-trees are used as they provide logarithmic time for common operations like insertion, deletion, sequential access and search. This COW-friendly B-tree in which the leaf node linkages are absent was originally proposed by Ohad Rodeh. In such trees, writes are never made in-place, instead, the modified data is written to a different location, and the corresponding metadata is updated. BtrFs, thus, has built-in support for snapshots, which are point-in-time copies of entire subvolumes, and rollbacks.

2.1 Data Structures

As seen in Figure 1, the BtrFs architecture consists of three data structures internally; namely `blockheader`, `key` and `item`. The `blockheader` contains checksums, file system specific `uuid`, the level at which the block

```
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;
}
```

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

```
struct btrfs _disk_key {
    __le64 object_id;
    u8 type;
    __le64 offset;
}
```

Figure 2: BtrFs Data Structures (src: BtrFs Design Wiki)

is present in the tree etc. The `key`, which defines the order in the tree, has the fields: `objectid`, `type` and `offset`. Each subvolume has its own set of object ids. The type field contains the kind of item it is, of which, the prominent ones are `inode_item`, `inode_ref`, `xattr_item`, `orphan_item`, `dir_log_item`, `dir_item`, `dir_index`, `extent_data`, `root_item`, `root_ref`, `extent_item`, `extent_data_ref`, `dev_item`, `chunk_item` etc. The `offset` field is dependent on the kind of item.

A typical leaf node consists of several items. `offset` and `size` tell us where to find the item in the leaf (relative to the start of the data area). The internal nodes contain `[key,blockpointer]` pairs whereas the leaf nodes contain a block header, array of fix sized items and the data field. Items and data grow towards each other. Typical data structures are shown in Figure 2.

2.2 Subvolumes and Snapshots

The BtrFs subvolume is a named B-tree that holds files and directories and is the smallest unit that can be snapshotted. A point-in-time copy of such a subvolume is called a snapshot. A reference count is associated with every subvolume which is incremented on snapshot creation. A snapshot stores the state of a subvolume and can be created almost instantly using the following command,

```
btrfs subvolume snapshot [-r] <source> [<dest>/]<name>
```

Such a snapshot occupies no disk space at creation. Snapshots can be used to backup subvolumes that can be used later for restore or recovery operations.

3 Send-Receive Code

Snapshots are primarily used for data backup and recovery. Considering the size of file system wide snapshots, detecting how the file system has changed between two given snapshots, manually, is a tedious task. Thus developing a clean mechanism to showcase differences between two snapshots becomes extremely important in snapshot management.

The `diff` command is used for differentiating any two files, but it uses text based[2] search which is time and computation intensive. Also, since it works on individual files, it does not give a list of files were modified/created/deleted on snapshot level. With the send/receive code, BtrFs can be instructed to calculate changes between the given snapshots and serialize them to a file. This file can later be replayed (on a BtrFs system) to regenerate one snapshot from another based on the instructions logged in the file.

As the name suggests, the send-receive code has a `send` side, that runs in kernel space and a `receive` side, which runs in user space. To calculate the difference between the two snapshots, the user simply gives a command line input given as follows:

```
btrfs send [-v] [-i <subvol>] [-p <parent>] <subvol>
```

`-v` : Enable verbose debug output. Each occurrence of this option increases the verbose level more.

`-i<subvol>` : Informs btrfs send that this subvolume, can be taken as 'clone source'. This can be used for incremental sends.

`-p<subvol>` : Disable automatic snaphot parent determination and use `<subvol>` as parent. This subvolume is also added to the list of 'clone sources'

`-f<outfile>` : Output is normally written to stdout. To write to a file, use this option. An alternative would be to use pipes.

Internally, this mechanism is implemented with the `BTRFS_IOC_SEND` ioctl() which compares two trees representing individual snapshots. This operation accepts a file descriptor representing a mounted volume and the subvolume ID corresponding to the snapshot of interest. It then calculates changes between the two given snapshots. The command sends the subvolume specified by `<subvol>` to `stdout`. By default, this will send the whole subvolume. The following are some more options for output generation:

- The operation can take a list of snapshot / subvolume IDs and generate a combined file for all of them. The parent snapshot can be specified explicitly. Thus, differences can be calculated with respect to a grandparent snapshot instead of a direct parent.
- The command also accepts 'clone sources' which are subvolumes that are expected to already exist on the receive side. Thus logging instructions for those subvolumes can be avoided and instead, only a 'clone' instruction can be sent. This reduces the size of the difference file.

The comparison works on the basis of meta-data. On detecting a metadata change, the respective trace is carried out and the corresponding instruction stream is generated. The output of the send code is an instruction stream consisting of `create/rename/link/write/clone/chmod/mkdir` instructions. For instance, consider that a new directory has been added to a file system whose snapshot has already been taken. If a new snapshot is then taken, the `send-receive` code will generate an instruction stream consisting of the instruction `mkdir`. The send receive code is thus more efficient as comparison is done only for changed files and not for the entire snapshots.

Taking into consideration the obvious advantages of using the send-receive code, the proposed solution uses it as a base for generating a `diff` between file system snapshots. To make the output of send code readable, we extract the data/stream sent from the send code and decode the stream of instructions into suitable format.

4 Design and Implementation

The proposed solution makes use of the BtrFs send ioctl `BTRFS_IOC_SEND` for diff generation. We have utilized the send-side of the send-receive code and extended the receive-side to give a view of the changes incurred to the file system between any two snapshots. This view

[2] `diff` does a line by line comparison of the given files, finds the groups of lines that vary, and reports each group of differing lines. It can report the differing lines in several formats, which serve different purposes.

```
Strategy: Go to the first items of both trees. Then do

If both trees are at level 0
  Compare keys of current items
    If left < right treat left item as new, advance left tree
      and repeat
    If left > right treat right item as deleted, advance right tree
      and repeat
    If left == right do deep compare of items, treat as changed if
      needed, advance both trees and repeat
If both trees are at the same level but not at level 0
  Compare keys of current nodes/leafs
    If left < right advance left tree and repeat
    If left > right advance right tree and repeat
    If left == right compare blockptrs of the next nodes/leafs
      If they match advance both trees but stay at the same level
        and repeat
      If they don't match advance both trees while allowing to go
        deeper and repeat
If tree levels are different
  Advance the tree that needs it and repeat

Advancing a tree means:
  If we are at level 0, try to go to the next slot. If
  that is not possible, go one level up and repeat. Stop when we found a level
  where we could go to the next slot. We may at this point be on a node or a
  leaf.

  If we are not at level 0 and not on shared tree blocks, go one level deeper.

  If we are not at level 0 and on shared tree blocks, go one slot to the right
  if possible or go up and right.
```

Figure 3: Tree Traversal Algorithm (src: BtrFs Send-Receive Code)

includes a list of the files and directories that underwent changes and also the file contents that changed between the two specified snapshots. Thus, a user would be able to view a file over a series of successive modifications.

The traversal algorithm is as given in Figure 3.

The BtrFs trees are ordered according to `btrfs_key` which contains the fields: `objectid`, `type` and `offset`. As seen in Figure 3, the comparison is based on this `key`. The 'right' tree is the old tree (before modification) and the 'left' tree is the new one (after modification). The comparison between left and right is actually a key comparison to check the ordering. When only one of the trees is advanced, the algorithm steps through to the next item and eventually one tree ends up at a later index than the other. The tree that reaches the end quicker, evidently, has missing entries which indicates a file deletion on this 'faster' side or a file creation on the 'slower'. The tree is advanced accordingly so that both the trees maintain almost the same level. The changes detected include changes in inodes, that is, addition and deletion of inodes, change in references to a file, change in extents and change in transaction ids that last touched a particular inode. Recording transaction IDs helps in maintaining file system consistency.

To represent the instruction stream in a clean way, we define the BtrFs `subvolume diff` command, as shown in Figure 4. This lists the files changed (added, modi-

```
btrfs subvolume diff [-p <snapshot1>] <snapshot2>

Output consists of:

I] A list of the files created, deleted
and modified (with corresponding number
of insertions and deletions).

II] Newly created files along with their
corresponding contents.

III] Files that have undergone write
modifications, with the corresponding
modified content in terms of blocks.
```

Figure 4: Btr-Diff and its output

fied, removed) between snapshot1 and snapshot2 where snapshot1 is optional. If snapshot1 is not specified, a `diff` between snapshot2 and its parent will be generated. The output generated by interpreting the data and executing the command above will be represented as given in Figure 4.

5 Performance and Optimization Achieved

Performance of Btr-diff was evaluated by varying the size of modifications done to a subvolume. Modifications of 1 GB to 4 GB were made and snapshots were taken at each stage. Btr-diff was then executed to generate a `diff` between these snapshots i.e. from 1–2 GB, 1–3 GB and so on. Benchmarking was done using the `time` command. The `time` command calculates the `real`, `sys` and `user` time of execution. The significance of each is as follows:

`real`: The elapsed real time between invocation and termination of the program. This is affected by the other processes and can be inconsistent.

`sys`: The system CPU time, i.e. the time taken by the program in the kernel mode.

`user`: The user CPU time, i.e. the time taken by the program in the user mode.

A combination of `sys` and `user` time is a good indicator of the tool's performance.

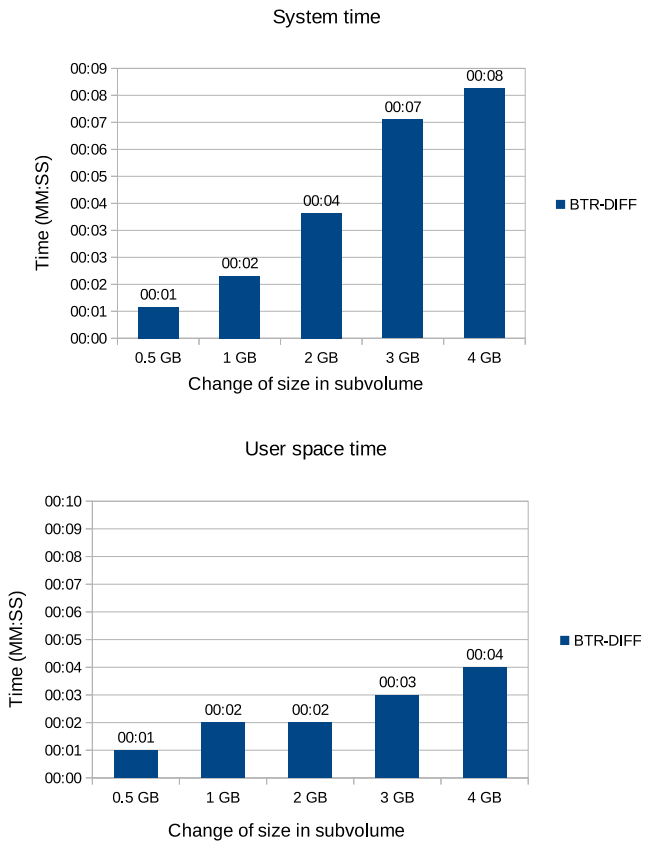

Figure 5: Time usage

The graphs in Figure 5 showcases the `sys` and `user` time performance of Btr-diff for different values of 'size differences' between the two given snapshots. The upper figure shows the time spent in kernel space by Btr-diff, while the lower figure shows the time spent in user space. The graphs clearly depict that the proposed method generates a `diff` in almost constant time. For changes of 1 GB, Btr-diff spends one second in the kernel and one second in user space. When these changes grow to 4 GB, Btr-diff maintains almost constant time and spends 8 seconds in the kernel and only 4 seconds in user space. Since the traversal is executed in kernel space, the switching between kernel and user space is reduced to a great extent. This has a direct implication on the performance.

There are various techniques/approaches that can be used for snapshot management. Using the send-receive based technique for BtrFs provides a lot of advantages over other generic comparison algorithms/techniques. Send/receive code is more efficient as its comparison is meant only for changed structure/files and not for entire snapshots. This reduces redundant comparisons.

6 Applications

Snapshot `diff` generation has several applications:

- Backups in various forms. Only the instruction stream can be stored and replayed at a later instant to regenerate subvolumes at the receiving side.
- File system monitoring. Changes made to the file system over a period of time can easily be viewed.
- A cron job could easily send a snapshot to a remote server on a regular basis, maintaining a mirror of a filesystem there.

7 Road Ahead

The possible optimizations/feature additions to Btr-diff could be as follows:

- Traversing snapshots and generating `diff` for a particular subvolume only.
- Btr-diff currently displays contents of modified files in terms of blocks. This output could be processed further and a `git-diff` like output can be generated which is more readable.

8 Conclusion

The lack of file system specific snapshot comparison tools for BtrFs have deterred the usage of its powerful snapshot capabilities to their full potential. By leveraging the in-kernel snapshot comparison algorithm (`send/receive`), a considerable reduction in the time taken for snapshot comparison is achieved. This is coupled with lower computation as well. The existing methods are generic and thus take longer than required to compute the diff. Our solution thus addresses this impending need for a tool that uses the BtrFs features to its advantage and gives optimum results.

References

[1] *Btrfs Main Page* http://btrfs.wiki.kernel.org/index.php/Main_Page

[2] Wikipedia - *Btrfs* http://en.wikipedia.org/wiki/Btrfs

[3] *Btrfs Design* https://btrfs.wiki.kernel.org/index.php/Btrfs_design#Snapshots_and_Subvolumes

[4] Linux Weekly News - *A short history of BtrFs* http://lwn.net/Articles/342892/

[5] IBM Research Report - *BTRFS:The Linux B-Tree Filesystem* http://domino.watson.ibm.com/library/CyberDig.nsf/1e4115aea78b6e7c85256b360066f0d4/6e1c5b6a1b6edd9885257a38006b6130!OpenDocument&Highlight=0,BTRFS

[6] Chris Mason - *Introduction to BtrFs* http://www.youtube.com/watch?v=ZW2E4WgPlzc

[7] Chris Mason - *BtrFs Filesystem: Status and New Features* http://video.linux.com/videos/btrfs-filesystem-status-and-new-features

[8] Avi Miller's BtrFs talk at LinuxConf AU http://www.youtube.com/watch?v=hxWuaozpe2I

[9] *Btrfs Data Structures* https://btrfs.wiki.kernel.org/index.php/Data_Structures

[10] Linux Weekly News - *Btrfs send/receive* http://lwn.net/Articles/506244/

[11] openSUSE - *Snapper* http://en.opensuse.org/Portal:Snapper