BTRFS: Investigation of several aspects regarding fragmentation

Elijah D. Mirecki
Athabasca University

February 26, 2014

Abstract

BTRFS defragmentation is currently an active topic of discussion among operating system designers. Defragmentation is especially important for BTRFS because of its copy-on-write (COW) nature, which causes data extents to be scattered all over the system when writing occurs.

This paper provides an investigation of the current BTRFS defragmentation algorithm as well as the autodefrag algorithm. Several illustrations have been created to represent how the defragmentation algorithm works at a high level. Discussion about possible solutions and ideas for defragmentation and autodefrag is also included.

The main problem with the current defragmentation algorithm is that it sometimes merges more extents than are needed to provide relatively unfragmented files. Another problem is that when choosing which files to defragment, flash drives and disk drives are treated the same. This is an issue because the two drive types should have different goals. Disk drives aim for contiguous space, while flash drives need less writes to prolong their lifetime.

The solutions and ideas discussed in this paper should theoretically allow defragmentation to merge extents more appropriately as well as meeting the goals for both flash and disk drives.

This paper should be followed by simulations comparing several areas of defragmentation and file selection algorithms. Some implementation details are provided as well in case the simulations return positive.

Several other filesystems are touched upon, including ext4, HFS+ and NTFS. Defragmentation and their fragmentation situation are both covered and compared briefly.

Introduction and background

Introduction to BTRFS

BTRFS is a extent based copy-on-write (COW) file system. An extent is basically an array of blocks. Instead of the filesystem metadata leaf nodes pointing at each individual block, only a single block pointer and an extent length must be set in the inode. With COW, when a block is modified, a write does not occur on an existing extent, but is written to a new extent on another part of the drive. Assuming a written block has occurred in the middle of an existing extent, the extent will have to be split into multiple parts. The inode will point to the beginning of the old extent, to the new extent created by the write, and then a third pointer to the end of the old extent. The data in the middle of the old extent may then be deallocated. This allows a much more versatile file system, but results in the system becoming much more fragmented.

Some of the main features of BTRFS include compression, clones, and snapshots. Snapshots allow the filesystem state to be saved for backups or archives. Clones are copies of files that initially share the same extents. As changes are made to the clone or the original file, the files will slowly share less with each other due to COW. Since BTRFS is a COW system, snapshots and clones are able to be created extremely quickly by simply creating new file pointers. Once there are no more pointers to an

extent on the drive, it will be deallocated.

BTRFS is also transaction based. Write operations are not completed immediately, but are placed in a transaction located in memory. The system commits all of the operations at regular intervals, or when otherwise required such as if the system is shutting down. This allows compression and write placement optimization to take place while the transaction resides in memory, before the data is even written to the disk. This is also a form of the delayed-allocation method – a method designed with the intent of reducing fragmentation by combining several write operations before the actual write occurs.

As mentioned above, BTRFS suffers greatly from fragmentation. For enterprise systems, the built-in BTRFS defragmenter may be run according to the system schedule. But for a general solution, BTRFS implements a mechanism called autodefrag. Autodefrag works by automatically choosing which files to defragment as they are modified. Files which are chosen will be defragmented along with each transaction being performed. This is a good solution for most systems because it keeps fragmentation low when small writes occur in files, which is the circumstance that generally causes the most file fragmentation.

Motivation for Research

As I was reading about ZFS on miscellaneous websites, I stumbled upon several other file systems such as BTRFS and XFS. At the time I started reading some more articles about BTRFS and the basic concepts; Copy-on-write, B-tree data structures, etc. Once it was required that I find a topic to research, my first thought was BTRFS, because I was quite interested with it in the past.

I then headed over to the Athabasca online library, where I found a recent article which covered many topics of BTRFS from several levels (Resource 1). As I was reading the article, it seemed as though the defragmentation system has been one of the largest problems with BTRFS currently. From what I have gathered in several articles, the autodefrag algorithm is the solution for selecting which files should be defragmented by default. Another problem that I realized was that the current defragmentation algorithm tends to strip out sharing of extents with clones and snapshots. This is a big issue, since clones and snapshots are core elements of the system. I also discovered that the defragmentation algorithms treat flash drives the same as disk drives. This stood out as yet another major problem, because flash drives behave much different. On a flash drive, access time is basically a constant time, so contiguous data doesn't matter nearly as much.

Problems

Excess Defragmentation.

The current defragmentation has a major flaw. When a file begins to be defragmented, the ENTIRE file will be scanned for extents that are worthy of being merged. There are several problems with this approach:

1. Performance will be affected.
2. A lot of sharing with be lost with clones and snapshots.
3. Drive lifetime will be reduced.

The performance will be degraded for several reasons. One reason is that the algorithm will have to scan the entire file, instead of just the part of the file that needs attention (the section of the file which has been modified). Another reason is of course, because extra extents will have to be COWed.

In a large file such as a database, this could be a massive overhead.

Much more sharing with snapshots and clones will also be lost unnecessarily. Defragmenting only the small section of a file which has been modified will most likely not require a lot of COW operations. If an entire file is defragmented, then extents all over the file will be merged, and therefore duplicated among clones and snapshots.

Drive lifetime will be decreased when more writing occurs. Drive lifetime is especially critical to flash drives, which are starting to dominate the secondary storage market these days.

Autodefrag Optimization for Flash Drives

Currently, the autodefrag algorithm basically just checks for the size of a COW operation to determine whether or not to add a file to the defragmentation list, despite which kind of drive is in use. The problem here is that with flash drives, defragmentation is not as big of an issue as on disk drives, so we don't want to defragment the file as often. The main problem is that flash drives have a shorter life span than disk drives, but don't provide a means of factoring this into account. Because the defragmentation algorithm will behave the same on a flash drive, it will try to merge extents to form contiguous data, which will wear out the flash drives if it is overdone. Another more minor problem is that performance may be impacted if more files are defragmented. The algorithm should be smarter to handle flash drive autodefrag differently to accommodate the factor of drive life span.

Methods

To begin exploring the circumstances of fragmentation in BTRFS, several documents and papers were read. The documents that were found covered a broad range of how BTRFS works, but were not specifically about the fragmentation issue or defragmentation algorithms, which were covered briefly. A general knowledge and understanding about how BTRFS works as a whole was gained from resource 1 mainly, but other miscellaneous articles and mailing lists were used as well.

Some of the main tactics used to explore the defragmentation algorithm were simply to poke around the Linux source code itself (version 3.12.8). The source code is a big place, so it could be difficult to find where to start. To find the starting point, the git logs for the Linux source code were searched regarding BTRFS defragmentation and autodefrag [5]. Once several functions were found that seemed to be reasonable starting points for exploring defragmentation and autodefreg, those functions were examined for comments and other method calls.

The Linux Cross Reference (LXR) [6] was used extensively throughout exploring the source code. Data structures and functions which were related to autodefrag and defragramentation were traced and examined.

Results and findings

Findings

The main starting places in the Linux source code that I found were the functions *btrfs_ioctl_defrag()*, *btrfs_defrag_file()* and *cow_file_range()*. The *cow_file_range()* function is what performs the COW operations. While performing these operations, it also checks if a file should be defragmented according to the autodefrag policy.

The current code will add a file to the list if a write is made that is under 64KB, and if the write is only part of the file. That is, if the beginning of the write is at the beginning of the file (position 0),

and the finishing point is beyond the EOF, it will not add the file to the defragmentation list. This makes sense, because why would we want to defragment a file with only one extent? Later on (line 858), some other code will attempt to write the extent inline with the leaf node. If the write is small enough, then this is possible, although if the extent is too large, a new extent must be created elsewhere. Below is the current code used to check for these conditions.

```
if (num_bytes < 64 * 1024 &&
            (start > 0 || end + 1 < BTRFS_I(inode)->disk_i_size))
      btrfs_add_inode_defrag(NULL, inode);
```

([2] - fs/btrfs/inode.c line 854)

As mentioned before, BTRFS is a transaction based system. Therefore, many writes to the same file should be completed in a single transaction, giving BTRFS the ability to optimize their placements. Whenever a transaction is committed, another operation that takes place besides writing to the disk is that the defragmentation list is crunched through to defragment all of the modified files before the changes are written to the disk. The default interval for committing is 30 seconds ([2] – fs/btrfs/ctree.h line 1994).

When a file begins to be defragmented (e.g. with the *btrfs_ioctl_defrag()* or *btrfs_defrag_file()* functions), a range structure - *struct btrfs_ioctl_defrag_range_args* - is configured to specify several parameters of the defragmentation operation. The parameters contained in this struct include the start and end positions of the defragmentation, compression type and the extent threshold. The start and end positions are used to determine which part of the file should be analyzed for defragmentation – Normally the entire file is included. The extent threshold, declared as *extent_thresh* is used to determine how large extents must be to be evaluated for defragmentation. For instance, if the threshold is set to 128KB, then any extents that are 128KB or over will NOT be merged with other extents. Setting the threshold variable to *1* will make the defragmentation operation include every extent, and setting it to *0* will use the default system threshold. The default threshold which will be used currently is 256KB ([2] - fs/btrfs/ioctl.c line 1193). The extent threshold is also another subject which the papers read did not explain, and in my opinion, is a vital part of the entire defragmentation algorithm. Without the extent threshold, the defragmentation algorithm would simply act as a file-copy defragmentation mechanism, which is an extremely wasetful use of resources for large files!

Examples and Illustrations

The following figures are abstract representations of extent allocation in files. In the figures, relationships are shown directly from the file/inode to the data extents. In reality, this is not the case. The actual structure of the extent allocation metadata in the filesystem uses pointers to headers in the extent tree from the inode. The headers then point to the actual extent data.

Moving on. Yellow blocks are files, blue blocks are extents that have not been changed, green blocks have been inserted, and cyan blocks have been moved/defragmented. In the cyan blocks, you will notice that there are '+' symbols. These are simply to denote which extents have been joined. Normally I would have placed both files on the same depth as well, but I thought there were too many arrows crossing that way, which was confusing, so I simply placed the cloned file on the bottom.

Below is a simple example of how extents could be shared despite new data being added or changed in one of the files which is sharing extents. In the example, a clone of File A is used, but the concepts of sharing are the same in both snapshots and clones.

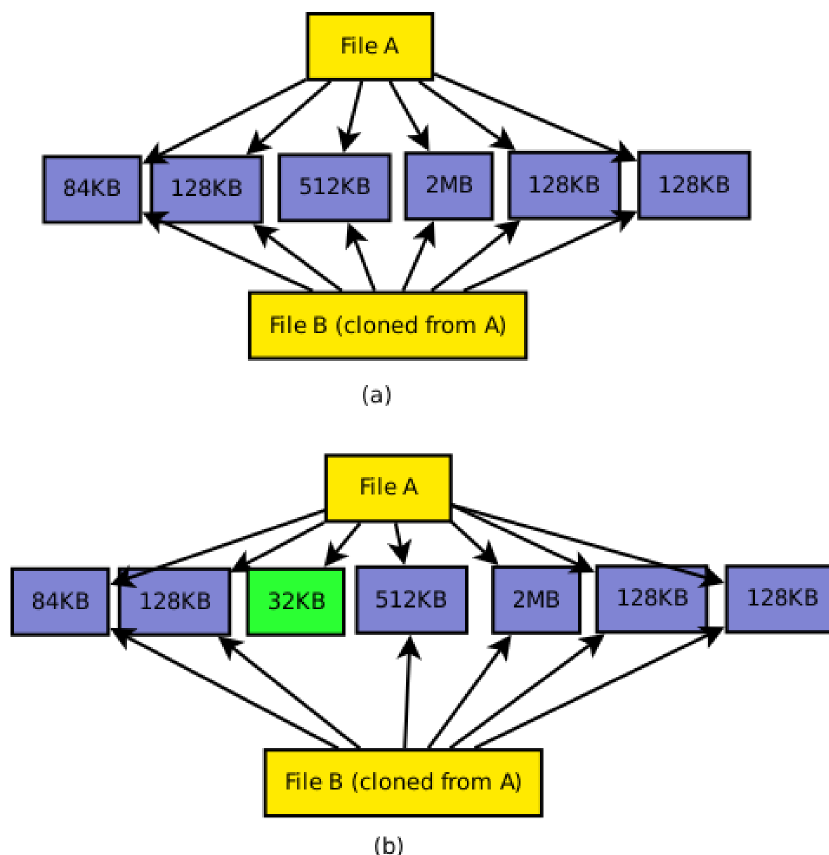

Figure 1. (a) Before a write (b) After a write – Added to the defragmentation queue.

The extent which has been written in the middle of the file is under 64KB, which according to the autodefrag policy, means that the file will need to be defragmented. Notice that even though a new extent has been added to File A, the old extents will still be shared with File B. This figure is a simple example, but the concepts should be generally the same despite the size and state of the file used as well as whether the operation which has been performed is a removal or simply a modification.

Once the file is in the defragmentation queue and the current transaction is committed, the real fun will begin. The simplest method to defragment is to just copy all of the extents to another location. This technique effectively merges all of the file extents into a contiguous extent – In this case, a single 3060KB extent. This is the way that several other file systems defragment files (covered in the Related Work section). There are several problems with this approach. First, in a filesystem which supports deduplication with snapshots and clones like BTRFS, all sharing will be lost with previous snapshots. This is obviously undesired behavior. Second, it is generally inefficient to move an entire file whenever it needs to be defragmented.

The way that BTRFS defragments files currently is only to defragment extents which are under a certain extent threshold (as mentioned above, the default for the threshold is currently 256KB). Each

group of contiguous extents that are under this threshold will be merged into a single extent. Below is an example of what the extent relationships would look like after the defragmentation is complete with the current algorithm.

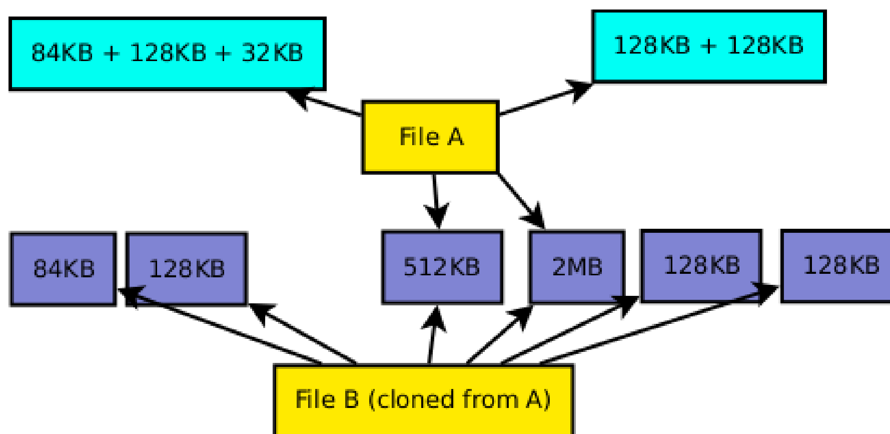

Figure 2. After the defragmentation is complete using the current algorithm.

In figure 2, notice how the 512KB and 2MB extents are still shared with File B. If File A is modified many more times only at the beginning area of the file, these extents should theoretically stay shared with File B forever. This algorithm is ideal for most circumstances because it both provides decent defragmentation and will share quite a bit of data with clones or snapshots.

Discussion

A big problem that I noticed with the current approach is that a file may not need so many extents to be defragmented. On the right side of Figure 2, notice that the two 128KB extents which were perfectly fine being separate, have been merged. Perhaps it is a good thing to merge the extents to gain increased performance, but the disadvantage is that those extents will no longer be shared with File B. The situation in this example isn't so bad, but imagine if File A had 100 128KB extents that were shared with File B. That would mean that the defragmentation algorithm would end up duplicating 12.5MB, which is clearly not very desirable.

Locality defragmentation. One idea for a modified approach is to merge only the extents that are adjacent to the modified extent which caused the defragmentation to be triggered. In this case, it would be the three extents on the left; 84KB, 128KB and 32KB. The two 128KB extents on the right would stay shared with File B. This approach may greatly enhance the performance of defragmentation of large files or files which have many small writes occurring in the same area. It will retain sharing of the parts of the file which are seldom modified with snapshots and clones. Imagine that there were 100 128KB extents on the right which were shared between File A and File B. All of these extents would stay shared between the files with the locality defragmentation algorithm, avoiding a lot of duplication.

Flash drives. On flash drives, the factors which determine a viable defragmentaion algorithm are important for different reasons than those of a hard disk drive. The access time of any data is basically always the same on a flash drive – O(1). Therefore, non-contiguous data won't affect the drive

performance as much as an HDD. The main issue with fragmentation on flash drives is the amount of extents and metadata. If many extents exist, more metadata (leaf nodes) will have to be used to reference the extent data. This not only causes wasted space from the extra metadata, but also degrades performance of the system due to the metadata needing to be accessed more times to find which extents the desired data is contained in. Another issue with flash drives is that they have a low amount of write cycles. Bits may only be written a certain amount of times before the drive will become immutable. This makes the life span on flash drives shorter than HDDs. The more that the defragmentation writes excessive amounts of data all over the drive, the more life will be sucked out of the flash drive.

The locality based defragmentation algorithm mentioned above would probably be extremely beneficial to flash drive defragmentation. Since this technique reduces the amount of COW operations performed while defragmenting a file, the life span of the drive should be prolonged.

Leaf count autodefrag. Autodefrag may possibly be improved as well to suit the needs of flash drives. Instead of simply checking for the size of the write to add the file to the defragmentation queue, what if the amount of metadata leafs was checked as well? If more than one leaf is being used at the time of a 64KB write being performed, then the file will be added to the defragmentation queue. Since the amount of leaf nodes is dependent on the amount of extents, the file will only be defragmented when the amount of extents is quite large. This would be a simple change to the autodefrag policy, yet it may increase the life span of flash drives drastically. Note that while this technique will enhance the performance of flash drives, it will actually decrease the performance of hard disk drives.

Related work

Ext4

Ext4 is a journaling file system which almost avoids the need to defragment altogether. Ext4 will still theoretically start to suffer from fragmentation after extensive use, but for most users, defragmentation is not necessary.

There are three main factors that cause ext4 to avoid fragmentation:

1. Extent based file allocation.

In the previous version (ext3), files were allocated to individual blocks, and each had to be referenced in the inode. On a 100MB file using a block size of 4KB, there would need to be 25600 pointers [4]! That isn't even the worst part. Because ext3 only writes single blocks at a time, it doesn't optimize the placement of them, so it ends up writing them in non-contiguous order, which results in fragmentation. A change in ext4 is the use of extent based file representation. A pointer and block amount pair may be used to represent each extent in the inode. Ext4 also writes entire extents at a time instead of searching just for possible single-block placements, as in ext3. Due to this, the filesystem will search for a contiguous chunk of many blocks to write to, which results in the disk being much less fragmented. In the example for ext3 above where 25600 pointers would be needed for the file, a single extent could theoretically be created using using a block amount of 25600. This technique is clearly much better than its predecessor.

2. Delayed allocation.

Ext4 doesn't actually write data as soon as a write call has been made. Instead, it stores all of the writes in memory in a buffer. When the file is flushed, that is when the real writing will occur. By

delaying the writes, the filesystem is able to perform several write operations into one single operation. The system may possibly even write multiple blocks into a single extent instead of splitting the writes up into multiple extents.

3. Padding around extents.

Another technique that ext4 uses is to keep some padding between extents. When a new extent is created, it isn't simply packed right back-to-back with the extents around it. Instead, some extra free space is left in it and the neighboring extents. This allows extents to expand in place instead of requiring the file to create a new extent on a different part of the disk allocated for the new data.

Despite these techniques to avoid fragmentation, after a long period of time with heavy use, ext4 will still become fragmented. The e4defrag program was created to defragment ext4.

File fragmentation [5] is the main type of fragmentation that defragmenters aim to correct. The algorithm which e4defrag uses is quite simple: to defragment a single file, it simply allocates a new extent on the disk which is large enough to store the entire file. It then copies all of the extents into a new single extent elsewhere on the drive.

File fragmentation isn't the only type of fragmentation, though. Another type is Relevant File Fragmentation [5]. This type of fragmentation occurs when files that are commonly accessed by the same program are scattered across the drive. The most likely circumstance is when files are located in the same directory, but are spread around the drive. The way that this type of fragmentation is corrected is by first defragmenting each file in a given directory, then moving all of the file extents into a contiguous group of extents. This will greatly improve access time in circumstances where many files in the same domain must be accessed often.

HFS+

HFS+ is a journaling extent based file system which is the default on OS X. The two main techniques that HFS+ uses to avoid fragmentation are on-the-fly defragmentation and delayed allocation, as in ext4.

HFS+ fragmentation avoidance mechanisms differ from ext4 when it comes to on-the-fly defragmentation. Whenever a file is opened, the file record (the equivalent of an inode) is tested to see how large it is and how many extents it contains to determine whether it should be defragmented or not. If the file is less than 20MB and contains more than 8 extents, then the file will be relocated to form a single contiguous extent. The reason 8 extents is the number used to determine whether to defragment or not is because a file record in HFS+ will only point directly to 8 extents. If any more extents are created, then one of the extent pointers in the file record must point to a B-tree to further reference more extents, which in turn slows down file operations.

HFS+, like ext4, is actually quite good at avoiding defragmentation, especially for small files. Large files will gradually get fragmented over time, but as in ext4, immense writing over a long period of time would have to occur for any serious performance deterioration.

NTFS

NTFS is the default file system for Windows. It is not a COW file system, but still greatly suffers from fragmentation.

In Windows XP, every file that was split into more than one piece was considered fragmented [3]. The problem with this technique is that if a file contains several very large pieces (e.g. over 128MB), the system will still try to defragment it, even though many of the pieces of the file should not be made contiguous. When defragmenting large pieces of files like that, more damage may be caused by trying to reorganize it than the damage caused by the actual fragmentation. One change of the algorithm in Windows Vista was to defragment pieces of a file which are larger than 64MB. The advantage with this scheme is that small files will be kept quite defragmented, but at the same time files with large chunks will retain those large chunks, and just defragment the smaller parts of the file. The disadvantage with this approach is that it doesn't provide a means of defragmenting SSD drives efficiently. Since SSD drives suffer from excessive writing, Windows will disable defragmentation by default if the drive is an SSD.

With Windows Vista and its predecessors, defragmentation routines were required to be run manually. Windows 7 introduced the use of an approach which automatically will defragment all of the files which have been modified at regular intervals. The interval in which this takes place could be set by the user and could vary from hourly to monthly.

Future Work and Conclusion

Future Work

Simulation. Theoretical speculation into the issues discussed is useful, but in the future, it would be efficacious to simulate them for performance and the resulting fragmentation of files. The results between the old and new techniques should be compared to determine if they are actually beneficial.

The locality defragmentation algorithm should be faster than defragmenting the entire file, in theory. The reasons for this are because less data will have to be COWed, and smaller sections of the file will have to be examined to decide which extents need to be merged, as discussed above. The future simulations should compare the locality defragmentation algorithm with the whole-file defragmentation algorithm. The test environments should include systems with many small files as well as systems with very large files such as databases.
Below are the main factors that should be examined:

- The amount of writes performed.
- The time it takes to complete defragmentation on the file.
- The amount of duplication caused.

Simulation should also be performed regarding the leaf-count based autodefrag condition for flash drives. The parameters that should be measured in this simulation should mainly include the amount of duplication and writes executed in clones of small files. Large files which include multiple leaf nodes to accommodate all of the extents should act the same as the current autodefrag algorithm.

Implementation. Below, some implementation ideas for the new ideas discussed are provided.

For the autodefrag addition, when a write occurs that is under 64KB on a flash drive, the amount of leaf nodes contained in the BTRFS inode should be tested. If multiple leafs are being used, then that is the only time the file should be added to the defragmentation list.

Pseudo code along with the current autodefrag conditions:

```
if (num_bytes < 64 * 1024 &&
            (start > 0 || end + 1 < BTRFS_I(inode)->disk_i_size) &&
            (!DRIVE_IS_FLASH || file_leaf_count(inode) > 1))
      btrfs_add_inode_defrag(NULL, inode)
```

([2] - fs/btrfs/inode.c line 854 - modified)

For the locality defragmentation, the starting point of the extent examination could start at the extent where the modification originates. The extents should be iterated backwards and forwards until an extent greater than the extent threshold is found. These are the only extents that should be considered for defragmentation. This may be implemented by setting the range with the *start* and *len* parameters in the *btrfs_ioctl_defrag_range_args* struct.

Pseudo code for the defragmentatation algorithm change:

```
struct btrfs_ioctl_defrag_range_args *range = /* Instantiate */;
struct btrfs_extent_item *modified_extent = /* Instantiate */;

struct btrfs_extent_item *last_extent;
struct btrfs_extent_item *first_extent;
struct btrfs_extent_item *previous_extent = modified_extent;
struct btrfs_extent_item *next_extent = modified_extent;

// Find the first extent that should be defragmented.
do {
        first_extent = previous_extent;
        previous_extent = get_previous_extent(first_extent);
} while (previous_extent != NULL &&
        extent_size(previous_extent) < range->extent_thresh);

// Find the last extent that should be defragmented.
do {
        last_extent = next_extent;
        next_extent = get_next_extent(last_extent);
} while (next_extent != NULL &&
        extent_size(next_extent) < range->extent_thresh);

// If the previous/next vars returned NULL, it is beginning/end of the file.
range->start = previous_extent == NULL
        ? 0
        : extent_position(first_extent);
range->len = next_extent == NULL
        ? -1 // When -1 is used as the length, it means unlimited length.
        : extent_position(last_extent) - range->start;
```

Conclusion

One of the main discoveries as a result of this project has been the extent threshold in the defragmentation algorithm. This is a critical part of the defragmentation algorithm, and without it, BTRFS defragmentation would simply copy entire files whenever defragmentation is needed, which is a terribly inefficient means of solving the fragmentation issue.

Locality defragmentation is a discussed technique to defragment only the section of the file in which a modification has occurred. The purpose of locality defragmentation is to avoid excessive merging of extents. This should both speed up the performance of the filesystem and reduce the amount of duplication caused by the defragmentation routine. Defragmenting entire files is much more computationally and drive intensive and it also forces extents that are currently shared with snapshots and clones to be COWed.

The leaf-count based autodefrag conditional for flash drives effectively determines whether a file should be added to the defragmentation list based on the amount of extents. As before, the size of the write is also a condition. This technique should improve the performance and efficiency of the entire defragmentation system in flash drives, which are commonplace today. Flash drives should experience a longer life time with these new techniques due to avoiding COW operations when reasonably possible.

Through the discussed conjectures of the BTRFS fragmentation situation, there are still many improvements that could be made. Several sections of the Linux source code have been pointed out and explained in this paper at a high level, which should aid future investigators.

References

[1] Ohad Rodeh, Josef Bacik, Chris Mason. August 2013. BTRFS: The Linux B-Tree Filesystem
ACM Transactions on Storage, Vol. 9, No. 3, Article 9
[2] Linux source code – 3.12.8
https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.12.8.tar.xz
[3] Steven Sinofsky. January 2009. Disk Defragmentation – Background and Engineering the Windows 7 Improvements
Engineering Windows 7
http://blogs.msdn.com/b/e7/archive/2009/01/25/disk-defragmentation-background-and-engineering-the-windows-7-improvements.aspx

[4] May 2011. Ext4.
http://kernelnewbies.org/Ext4
[5] Akira Fujita. September 2010. Outline of Ext4 File System & Ext4 Online Defragmentation Foresight
NEC Software Tohoku, Ltd.
http://events.linuxfoundation.org/slides/2010/linuxcon_japan/linuxcon_jp2010_fujita.pdf
Mark Russinovich. 1997. Inside Windows NT Disk Defragmenting
http://mirrors.arcadecontrols.com/www.sysinternals.com/Information/DiskDefragmenting.html
Ohad Rodeh. February 2008. B-trees, Shadowing, and Clones
ACM Transactions on Storage, Vol. 3, No. 4, Article 15
Amit Singh. May 2004. Fragmentation in HFS Plus Volumes
Mac OS X Internals
http://osxbook.com/software/hfsdebug/fragmentation.html
Laura LeGault. February 2009. HFS+: The Mac OS X File System
http://pages.cs.wisc.edu/~legault/miniproj-736.pdf
Valerie Aurora, July 22, 2009. A Short History of BTRFS
http://lwn.net/Articles/342892/
Mailing list archives
http://blog.gmane.org/gmane.comp.file-systems.btrfs
http://www.spinics.net/lists/linux-btrfs/msg31028.html
http://comments.gmane.org/gmane.os.solaris.opensolaris.zfs/48392
Linux kernel source tree: Git commit logs regarding defragmentation
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4cb5300bc839b8a943eb19c9f27f25470e22d0ca
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6702ed490ca0bb44e17131818a5a18b773957c5a
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=e9d0b13b5bbb58c9b840e407a8d181442f799966
The Linux Cross Reference (LXR)
http://lxr.linux.no/, http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/