RJ10465 REVISED (A1004-013) April 26, 2010
Computer Science

IBM Research Report

Defragmentation Mechanisms for Copy-on-Write File-systems

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

Defragmentation Mechanisms for Copy-On-Write File-systems

Ohad Rodeh

April 23, 2010

Abstract

File systems like ZFS, WAFL, and BTRFS use a copy-on-write transactional model to implement updates, crash recovery, snapshots, and clones. The resulting disk layout is one where there could be many physical pointers to a single block. This allows quick lookup when accessing a block from all the snapshots is belongs to.

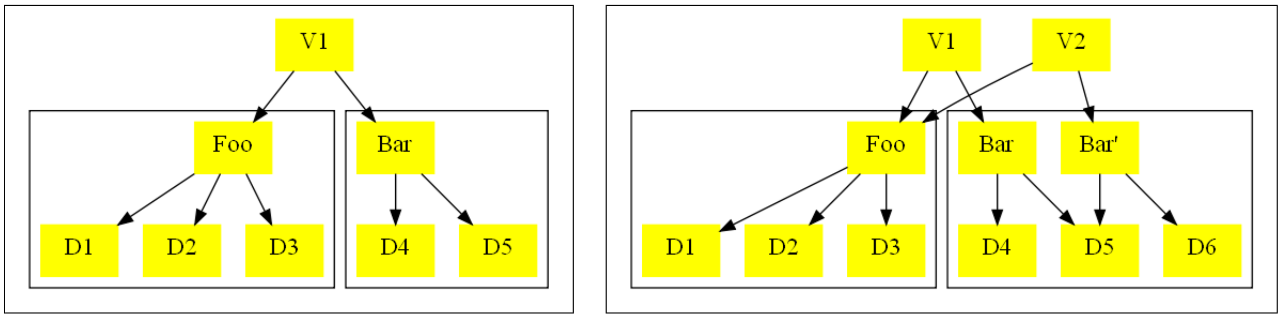
When performing defragmentation, blocks are moved, requiring the update of all incoming pointers. A reverse lookup capability is needed, allowing the quick location of all pointers to a particular block. One solution is to track back-references. However, this adds an additional meta-data structure that needs to be kept up-to-date, taking up disk space and memory.

This document suggests a different approach, applying ideas from the realm of programming language heap compaction to file-system defragmentation.

1 Introduction

File systems like ZFS [9], WAFL [3, 4], and BTRFS [7], use a copy-on-write update model. This facilitates the implementation of transactional consistency, crash recovery, snapshots, and clones. The file-system on-disk layout is equivalent to a directed-acyclic-graph (DAG), with physical pointers between blocks. The roots of the graph are the snapshots and clones, the middle nodes are indirect blocks, and the bottom nodes are data blocks.

Figure 1(a) shows an example of a volume (V_1) with two files, Foo and Bar. Figure 1(b) depicts sharing between V_1 and volume V_2. Foo is shared in its entirety, while Bar has been modified in V_2, and only one of its blocks (D_5) is shared.



(a) A file-system tree (b) Sharing with a snapshot

Figure 1: An example of a file-system with clones. (a) shows a file system volume, V_1, with two files: Foo and Bar. (b) shows sharing between volumes V_1 and V_2.

There are cases when blocks must be moved. For example, file-system defragmentation, and file-system shrink. When there is a bad-block, it is desirable to figure out which files it belongs to. One method to support these use cases is to track back-references [8]. The drawback is that an additional heavy weight meta-data structure is needed, taking up disk-space, memory, and requiring constant updates.

In this work we focus on the defragmentation problem, and suggest a different approach. Programming languages such as Java and C# employ a garbage collector that, among other things, performs heap compaction. The heap compaction algorithm cannot be applied directly to the file-system case because the file-system connectivity graph does not fit in memory. We use an external memory approach, and construct a defragmentation algorithm that can make do with a limited amount of memory.

This paper is organized as follows: Section 2 described related work on defragmentation. Section 3 describes the simplest case, how to move a file-system onto a new empty disk. Section 4 addresses compaction in place. Section 5 deals with updates occurring while blocks are being moved. Section 6 shows how to split the work into small, manageable pieces. Section 7 describes resource consumption. Section 8 summarizes, and section 9 contains acknowledgments.

2 Related work

Log structured file-systems [5, 6] use a form of copying garbage collection in order to free up space. They manage disk space in chunks such that writing a full chunk is efficient in terms of sequential disk IO. Reasonable chunk size at the time of writing is 1-4MB. New data is collected and written out in complete chunks. A copy-on-write model is used, old data is never overwritten. In order to release space, several chunks are chosen, live reachable blocks are collected, and written out in bulk to free chunks. The old chunks can then be reused. This scheme requires figuring out which file-system blocks are still alive at the time of collection. The method is adding back-references into the beginning of a chunk. This is difficult with snapshots, because one must find *all* pointers into a block in order to safely move it. One method described in the literature [2] is adding a level of indirection so that logical block addresses used by the file-system do not change, allowing moving blocks on disk during collection.

WAFL [3] manages disk space in fixed sized blocks, where a typical size is 4KB. Updates are done in copy-on-write fashion, old data is never overwritten. An individual volume in the file-system is essentially a tree. The root points to the i-node file, which points to indirect blocks, which point to data blocks. The creation of volume snapshots is supported. A snapshot can be writeable, in which case it is also called a *clone*. The snapshot operation is space-efficient, unmodified blocks are shared, as much as possible, between the original volume and its snapshot. This means that the file-system as a whole is a Directed-Acyclic-Graph (DAG) where the root nodes are the snapshot roots, and the bottom nodes are the data blocks. In [4], an additional indirection layer was added to WAFL. This, in theory at least, allows the relocation of blocks at the disk level without any modifications to higher level file-system pointers.

ZFS [9, 1] is a copy-on-write file-system developed for the Solaris operating system by SUN^{TM}. It supports advanced features like snapshots, clones, checksumming, deduplication, and RAID. Much like WAFL, its file-system graph looks like a DAG. Unlike WAFL, ZFS does not have an indirection layer that allows transparently moving blocks. This means that in order to move a block, all of its ancestors must be found, and then updated. A feature that will do just that is called the *Block Pointer rewrite*, and it is on the ZFS roadmap.

BTRFS [7] is a copy-on-write Linux file-system developed by Oracle^{TM}. It supports advanced features like snapshots, clones, and checksumming. Metadata is allocated in fixed sized blocks, while data is allocated in variable sized areas called extents. Typical values at the time of writing are 4KB for a block, and a multiple of 4KB for an extent. BTRFS shares the DAG graph structure with ZFS and WAFL. There is no layer of indirection allowing blocks to reallocated behind the scenes, instead, back-references are supported. Each block and extent have a back pointer(s) to their ancestor(s). This allows moving individual extents when needed.

It is possible to generically track back-references [8]. Such an approach can be tacked on to many file-systems, allowing them to move blocks when necessary.

Existing approaches either use an additional indirection layer, or back-references. Indirection layers add overhead to the common operation of pointer dereference, and the metadata required for the mapping competes for memory with other file-system data structures. Back-references add an additional on-disk metadata structure that needs to be kept up to date consistently with the rest of the file-system. This work attempts to take a third route, and use ideas from heap compaction instead.

3 Compact-move

The simplest compaction case is when a filesystem needs to migrate from one disk to another. This might happen, for example, if the old disk is performing badly, or if it damaged. The filesystem is migrated to the new, empty, disk and compacted. This is called the *compact-move* case.

Compaction works at the graph level, it starts at the root and traverses the file-system graph. It can tell what are pointers and what is data. The assumption is that the file-system does not use hard-coded disk addresses that cannot be moved, and that are not visible to graph traversal. Further, all pointers reside in indirect, meta-data blocks, and bulk data resides in extents that can be variable size. Only meta-data blocks containing pointers need to be traversed.

Compaction is split into several phases:

Discovery: find the independent moveable extents, and store them in an in-memory tree structure called the *x-tree*.

Decision: decide which extents to move, and where to move them. Update the x-tree with the destinations.

Movement: move extents and fix pointers.

Graph traversal is a key sub-algorithm, it is performed by starting at the roots and using breadth-first-search. Note that dead blocks will never be visited, and that data blocks are never traversed. Traversal stops once no more new blocks can be reached.

The discovery phase performs a graph traversal and records any extents visited in a structure called an *x-tree*. Information such as the snapshot, i-node, and file-range are recorded for each extent, this will be used later to make allocation decisions. It is possible for extents to intersect, in which case they are merged. Figure 2 shows an example graph fragment. Figure 3 shows the extents recorded for this fragment. Extents $D_{10} - D_{15}$ and $D_{15} - D_{16}$ overlap on block 15 and they are merged. Extent $D_{22} - D_{28}$ is included in $D_{20} - D_{30}$, and it is not recorded independently.

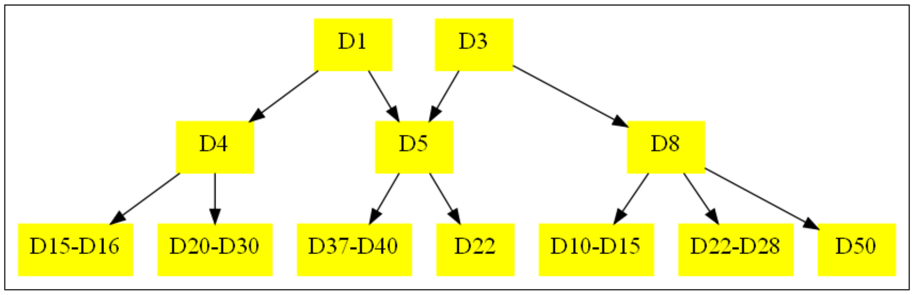


Figure 2: A graph fragment. Each node is denoted by the letter "D" and its block address, or block range. D_1 and D_3 are individual blocks. $D_{20} - D_{3o}$, and $D_{37} - D_{40}$ are extents.

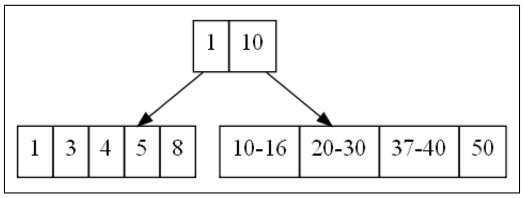


Figure 3: The independent extents for the graph fragment from Figure 2. For example, extents $D_{10} - D_{15}$ and $D_{15} - D_{16}$ overlap on block 15 and they are merged. The set of extents is recorded at the leaf nodes, the intermediate nodes serve to build a search tree for efficient access.

At the end of the discovery phase, we have a set of live extents that can be moved independently of each other, as long as references are correctly updated. These are also called *components*. The maximal number of components is equal to the number blocks in the file-system. This can occur only if the file-system is maximally used, and if the allocator could find no extents, only individual blocks.

In the decision phase target locations for each component are determined and recorded in the x-tree. Various policies and optimization goals can be used. Example targets are (1) co-locating contiguous data ranges in a file, (2) related files, or (3) data from a particular snapshot. Data placement determines how efficient an access pattern is going to be.

A simplistic algorithm that co-locates contiguous file data can traverse the x-tree and sort components lexicographically according to snapshot, i-node, and file-range.

The general problem of choosing an optimal goal, and an allocation algorithm to implement it, is beyond the scope of this work. Here, we focus on the defragmentation mechanisms, and assume that a policy choice has been made.

The decision algorithm traverses the in-memory x-tree, reads the per component information, and records target locations. This does not require any disk IO, it may require additional memory, for example to sort the x-tree.

In the Movement phase the file-system graph is traversed. Each data extent is copied to its target. Each meta-data block is scanned for references, these are fixed to point to the new locations, and the modified meta-data block is written to its target location.

At the end of these stages, the new disk holds the new image of the file-system, the old disk can be removed. Since all data is written into a new disk, crash recovery is simple. Upon crash, the updates are lost.

4 Compact-in-place

Compacting a file-system in place is challenging when it is heavily used, and there is little room in which to move extents. An additional challenge is that meta-data blocks need to be updated in place, which goes against the grain of copy-on-write file systems.

The algorithm runs along the same line as the compact-move algorithm. The Decision phase does not have the freedom to reorder and move all the data. Therefore, more limited goals must be set for it. For example, to compact all the data into the beginning of the address space, without trying to locate data from the same file close together. The x-tree records target locations for each component, if a component will not move then its target location is the same as its original location.

The Movement phase traverses the file-system DAG. There are cases where meta-data blocks do not move, but their pointers are updated. Since compaction is performed in place, there may not be enough space to write the meta-data (and all paths leading to it) in an alternate location. The upshot is that such meta-data blocks will be updated in place, contrary to the file-system strategy of copy-on-write.

In order to maintain crash-consistency, the entire compaction operation must be treated as one large transaction. All updates in-place must be logged to disk. In addition, the entire address translation x-tree must be logged to disk. When compaction completes, the log can be erased, and the space taken by old versions of migrated extents can be released.

5 Updates during compaction

Compaction takes up a significant amount of time, and IO cannot be halted for the duration.

While data movement is going on, some references will point to a target location, where the data has not been copied there yet. In other words, some new references will point to garbage. In order to circumvent this problem, all pointer de-references go through a reverse-mapping x-tree (r-x-tree), that maps all new addresses to their old locations. Reads from new addresses will be converted into reads from old addresses. The old data is immutable, has not been erased, and any pointers it contains are still valid. Therefore, accessing old disk addresses is correct, for the duration of compaction.

At the end of compaction all new addresses will be valid; the x-tree, r-x-tree, and the old data will be erased. The r-x-tree provides the freedom to move extents independently, in any order, allowing optimization of the IO schedule.

New writes are performed using copy-on-write, using the standard file-system methodology. Therefore, new data will never overwrite old data. Since old extents are being moved, new extents must be allocated in areas not used by compaction. Many new indirect blocks will have pointers to older extents. These references must employ the new extent locations, where extents will land at the end of compaction.

6 Incremental compaction

Compaction is a background operation that goes on while the file-system is in operation. Only limited memory and IO bandwidth are available to it. The upshot is *incremental compaction*, running compaction on a portion of the file-system at a time.

In order to limit the effort to a particular on-disk area, the x-tree is restricted to extents that lie entirely within that range. Extents outside the range will not be moved.

An additional constraint is memory. The access pattern to the x-tree is random, and keeping it memory resident is crucial for performance. Therefore, the area being compacted must be limited, such that its x-tree is within the memory budget.

7 Resource consumption

This section gives estimates for the amount of resources needed to run compaction. It is assumed that block addresses are 64bits, extent lengths are 32bits, i-nodes are 32bits, and snapshot ids are 32bits. Some of the calculations involve counting bytes, these are marked by "B". For example $5B$ means 5 bytes.

Compaction requires in-memory meta data per component. The x-tree maintains the following record for each component:

Field		**Size**
Has the component already been moved	flag	1 bit
Original location on disk	offset	8 bytes
Target location on disk	offset	8 bytes
Length		4 bytes
Decision support information	snapshot, i-node, file-offset	16 bytes
Total size		40bytes

The r-x-tree is used while data is being moved. It includes a mapping from new to old locations. The per component record is:

Field		**Size**
New location on disk	offset	8 bytes
Original location on disk	offset	8 bytes
Length		4 bytes
Total size		20bytes

We approximate space usage of a search tree to be twice the size of its keys. For example, a binary tree with n records has $2n - 1$ nodes. A b-tree with n records has nodes that are 50% full on average. Both the x-tree and the r-x-tree are search trees, combined, they require: $2(40 + 20)B \times \#components$.

Traversal of the indirect-block graph is done in breadth-first-search (BFS). This requires keeping track of levels i and $i + 1$ of the DAG. A search tree that keeps track of block-addresses is needed per level. The total amount of addresses in levels i and $i + 1$ is bounded by the total number of indirect blocks in the file-system. Therefore, the trees takes up no more than $2 \times 8B\times$ *#indirect blocks*.

All the live indirect blocks are traversed twice. Once during Discovery, and again during Movement. The indirect blocks may overrun the memory budget for the compaction operation, therefore, an external memory algorithm is required.

Filesystem indirect blocks are wide, and have room for a large number of pointers (at least 100). This makes the file-system DAG shallow. Therefore, our strategy is to read the DAG in layers and use our available memory budget for the top levels of the DAG. In other words, we traverse the DAG using breadth-first-search (BFS). While at layer i, we make a list of all disk addresses of level $i+1$, and read them in batch, sorted in disk order. If we can keep in memory the entire DAG except for the lowest layer, then a single parallel disk read would suffice in order to complete a traversal of the indirect blocks. Since two traversals are required, we would end up with two parallel reads.

The file-system comprises of snapshot and volume trees, all having different depth. In order to allow reading the DAG in layers, the file-system must tag each indirect block with a depth marker. BTRFS for example, does just that. Without such markers, the BFS traversal blindly goes down the tree, reading nodes from different depths, and filling memory with deep nodes that are of less value.

The memory budget is split as follows:

Use	**Memory size**
x-tree and r-x-tree	$120B \times \#components$
Cache for the top layers of the metadata DAG	-
Addresses for levels i and $i+1$ of the DAG	$16B \times \#indirect\ blocks$

The disk IOs are split as follows:

Reading the top k layers of the DAG	once
Reading the bottom layers of the DAG	twice
Reading and writing all migrated blocks	once

It is interesting to see how much IO and memory would be needed for a real example. Examine a case where:

1. File system size is 1TB, it is 65% utilized
2. Block size is 4KB
3. Live indirect blocks account for 0.5% of the space
4. Component size is 16KB on average
5. Compaction is run on a 1GB area at a time

Therefore:

1. The number of components in the area to be compacted is $\frac{1GB}{16KB} = 64K$
2. The number of indirect blocks is $\frac{1TB}{4KB} \times 0.005 = 256M \times 0.005 = 1.25M$
3. The total amount of space taken by indirect blocks is $1TB \times 0.005 = 5GB$

Assuming that the number of blocks increases by a factor of 20 between levels i and $i+1$ of the file-system DAG, we get that $\frac{5GB}{400} \sim \frac{5\times10^9B}{400} = 1.25 \times 10^7B = 12.5MB$ are needed to cache all but the bottom two DAG levels.

The memory budget is split as follows:

Use	**Memory size**
x-tree and r-x-tree	$8MB$
Cache for the top layers of the metadata DAG	$12.5MB$
Addresses for levels i and $i+1$ of the DAG	$22.5MB$
Total	$45MB$

The IO is split as follows:

Use	**Amount of IO**
Reading the top of the DAG into memory	$12.5MB$
Reading the bottom two layers of the DAG, twice	$2 \times 5GB = 10GB$
Reading and writing all live blocks in the compacted area	$2 \times 65\% \times GB = 1.66GB$
Total	$12GB$

8 Summary

Defragmentation is an important problem for long running file-systems. File systems that use copy-on-write have a particularly difficult time performing defragmentation because they allow multiple physical pointers into the same block. This work has shown how to compact such file systems without using additional on-disk structures, and while limiting resource use.

9 Acknowledgments

The author would like to thank Chris Mason for describing the way BTRFS implements back-references, and general discussions on the subject.

Thanks is also due to Mark Hayden who helped review this paper, significantly improving its quality.

References

[1] J. Bonwick. `http://en.wikipedia.org/wiki/ZFS`.

[2] C. Soules, G. Goodson, J. Strunk, and G Ganger. Metadata Efficiency in a Comprehensive Versioning File System. In *USENIX Conference on File and Storage Technologies (FAST)*, 2003.

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

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

[5] J. Ousterhout and F. Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. In *ACM SIGOPS*, January 1989.

[6] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

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

[8] P. Macko, M. Seltzer, and K. A. Smith. Tracking Back References in a Write-Anywhere File System. In *File and Storage Technologies (FAST)*, 2010.

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