RebFS: A High-Performance Versioning File System

Eric Lubin
eblubin@mit.edu
R07 T1, Adam Chlipala
6.033 Design Project 1
March 22nd, 2013

1 Introduction

The Resident Bit File System (RebFS) is a modified version of the UNIX file system that maintains the version history of files in a space-efficient manner. It allows for automatic garbage collection, exclusion from versioning, and searching across old versions of a file. To accomplish this, the file system creates a new inode for every version of a file and references common blocks from multiple versions of the same file.

1.1 Design Overview

Versioning information is stored within version sets that are maintained for every versioned file on the file system. Old versions may be accessed or listed using a path name convention discussed below. Files may be excluded or included from versioning on a per file basis, and entire subsets of directories may be excluded all at once. RebFS allows blocks to be referenced by multiple versions of the same versioned file by maintaining a resident bit in each block pointer.

To make sure that versioned files are correctly created, maintained, and deleted, RebFS modifies the default Unix implementation of `write()`, `open()`, and `read()`. Additionally, the system adds a `Set_Versioned()` system call to customize which files and directories should be included in automatic versioning.

1.2 Trade-Offs and Design Decisions

A significant design decision made in RebFS is to store every version in a unique inode and allow multiple inodes to reference the same block. Complicating the manner by which blocks are stored in the file system requires a comprehensive set of rules for block reference counting. However, the reference counting remains transparent to the system when reading files. An alternative diff-based approach that describes only the changed blocks and their locations might be simpler in implementation, but the high performance cost when reading old versions of files is unacceptable for this design.

For block reference counting, the resident bit approach used in RebFS requires an additional bit to be stored for each block pointer in the file system. The usage of the high-order bit shortens the address space of accessible blocks by a factor of two. To accommodate for this, we double the size of every block in the file system to make the entire disk accessible with one less bit. The notion of resident and nonresident blocks extends directly to the indirect block paradigm as well.

2 File System Design

RebFS uses *version sets* to store information about all versions of a file. We allow multiple versions of a file to reference the same blocks to minimize the additional amount of disk space required for versioning.

2.1 Versioning Hierarchy

On the highest level, we organize a file and all of its past versions into a new data structure called a version set. Each version set keeps a pointer to the current version of the file. We support version sets by adding a new type of inode to the existing Unix file system. Analogous to those of a directory, the blocks of an inode of this type contain a list of pointers to other inodes that represent different versions of the same file (Figure 1). The `read()` system call will forward all attempts to read a versioned file to the current version, or *head*, of the file. This behavior allows the versioning system to remain transparent to the user and all other applications, except when explicitly called.

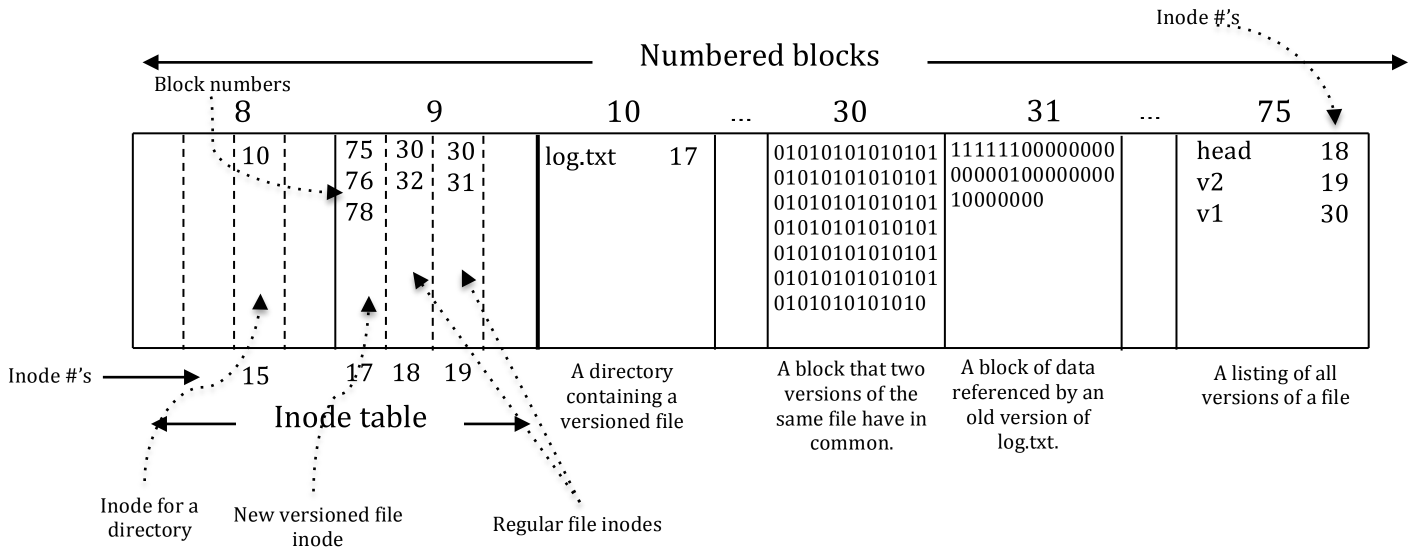

Figure 1. Shows the directory-like behavior of versioned files as well as how blocks may be referenced from multiple versions of the same file.

2.2 Accessing Previous Versions

Versioning info is built directly into the path name layer. This strategy minimizes the complexity of adding additional system calls. The user and other applications may retrieve old versions and list all versions of a particular file using the following set of conventions.

- `open(<path_to_versioned_file>,flags)` – opens the versioned file for reading and/or writing. All reads will get forwarded to the *head*, as mentioned above. All writes follow the automatic versioning pattern described in section 2.4.3.
- `open(<path_to_versioned_file>/,READ)` – lists all previous versions of the versioned file. The system opens the versioned file inode as it would a directory. Writing to this virtual directory is unsupported.
- `open(<path_to_versioned_file>/v<version_number>,READ)` – opens a read-only previous version of the versioned file.

To support these operations, we modify the `Lookup()` system call discussed in section 2.5 of the textbook. `Lookup()` must now be allowed to look for files in both directories and versioned file inodes.

2.3 Excluding Files from Versioning

RebFS contains a set of user configurable rules to determine which files will be versioned. By default, all user files, but not system files, are enabled for versioning. All inodes that are the versioned file type are automatically versioned. All inodes that are the regular file type are not versioned. We add one new system call, `Set_Versioned(path,versioned)`, to give the user control over setting a file to a particular type. The pseudocode for this new system call is included in Figure 2 below.

```
1   procedure Set_Versioned(path_name,versioned):
2      if Traverses_Versioned_File(path_name) return FAILURE
3      //do not allow user to version a version
4      //(i.e. Set_Versioned(~/file.txt/v4,True))
5      inode_number = GeneralPath_To_Inode_Number(path_name)
6      inode = Inode_Number_To_Inode(inode_number)
7      if inode.type = DIRECTORY return FAILURE //directories cannot be versioned
8      if inode.type = VERSIONED_FILE
9         if versioned = True return
10        //delete all old versions of the file
11        //delete the versioned file inode
12        //move the head to the `inode_number` location in the inode table
13     if inode.type = FILE
14        if versioned = False return
15        //move the inode to a new empty location in the inode table
16        //create a new versioned file inode at the `inode_number` location
17        //hard links referencing 'inode' will now reference the versioned file
18        //set 'inode' to the head of the versioned file, with all resident bits = 1
```

Figure 2. Pseudocode depicting the implementation details of the Set_Versioned() system call that converts a file between the versioned and unversioned states. We shuffle inode numbers to verify hard links are not broken.

We must be careful not to break hard links when creating and deleting versions. We do not allow hard links to reference inodes within a version set. Users are still permitted to make hard links to the versioned file inode. The behavior of hard links for unversioned files remains unchanged.

All files within a directory may also be excluded from versioning. We add a bit to each inode, only relevant for directories, that stores this information. When a versioned file is opened, we add an additional bit to the file descriptor table that indicates whether any of its parent directories have been disabled for versioning. If so, no new versions are created. When we exclude a directory from versioning, we prompt the user whether all contained versioned files should be converted to regular files.

2.4 Ensuring Space Efficiency

The cornerstone of the versioning file system is its storage of only the changed blocks of a file. When dealing with large files, it is critical not to have to copy thousands of blocks every time a new version is created. For this reason, we allow a block to be referenced more than once by different versions of the same file.

2.4.1 Block Reference Counting

To decide when blocks should be marked as free in the system bitmap, we augment the 32-bit block pointer in versioned files with a high-order resident bit. For a given physical block on disk, only the most recent version of a file that contains this block will have the resident bit set in its pointer to the block. If a block's pointer is set to nonresident, there must be a newer version referencing the block. An example is shown in Figure 3 below.

head	A1	B3	C3	D2		
v4	~~A1~~	B2	~~C3~~	~~D2~~	E2	F1
v3	~~A1~~	~~B2~~	C2	~~D2~~	~~E2~~	~~F1~~
v2	~~A1~~	~~B2~~	C1	D1	E1	
v1	~~A1~~	B1	~~C1~~	~~D1~~		

Newer Versions ↑

Key: Resident bit = 1 ~~Resident bit = 0~~

Figure 3. An abstracted version of the file system showing resident bit values in a version set. Each row represents a different version of the file. Each cell represents a block pointer to a physical block. Numbering of the blocks denotes specific changes in a block across versions. An empty cell indicates the end of the file.

2.4.2 Garbage Collection

We allow deletion of a version as long as all older versions of the same file will be deleted as well. To delete, we iterate over the blocks of a version and mark the block as free in the system bitmap if the pointer's resident bit was set to 1. If an indirect block is nonresident, we do not explore its children because they cannot be resident to this version.

Automatic garbage collection is performed based on user configurable metrics that are checked before new versions are created. For example, the user may specify that when a version set is taking up 100 times more space than its *head*, delete as many old versions as needed so that the resulting version set takes only 50 times more space than the *head*. Rules can also be made to delete versions when they get too old. In addition, when the disk gets close to full, we clear up space by deleting all versions older than a specific date.

2.4.3 Automatic Versioning

Using a variation on the copy-on-write mechanism, we automatically version all files that are not disabled from versioning, either explicitly or through a parent directory. At most one new version of a file will be created between `open()` and `close()` of a file, regardless of how many writes. To achieve this, the file descriptor table stores an additional `has_copied` bit for versioned files opened with write flags. When `write()` is called on a versioned file we:

1. Check `has_copied`. If it is 0, make a new version of the file with the same block pointers as the *head*, but with all resident bits set to 0. This new inode represents the newest version other than the *head*. Set `has_copied` to 1.

2. Write the new block to an empty location on the disk.
3. Update the block pointer in the *head* at the file descriptor's `offset` to this new location.
4. Set the resident bit to 1 for the block pointer at index `offset` in the inode created in step 1.

We demonstrate the versioning process using the example in Figure 3 above. Assume that there are versions v1, v2, v3, and *head* and that the top row of the table is about to be written to disk. The steps are outlined below.

1. `write(fd, 'B3')`
   a. Creates a new version, v4, with all the same block pointers as the *head* but all resident bits set to 0.
   b. Sets the `has_copied` bit to 1 in the file descriptor table.
   c. Writes the block 'B3' to an empty location on disk.
   d. Changes the B block pointer in *head* to point to 'B3'.
   e. Updates the B block pointer in v4 to be resident.
2. `lseek(fd, BLOCK_SIZE*2, SEEK_CUR)`
   a. Changes `offset` in the file descriptor table so the next write location is block E.
3. `write(fd, END_OF_FILE)`
   a. Reads `has_copied` = 1 and does not create a new version.
   b. Sets the E block pointer to 0, indicating the end of the file.
   c. Updates the block pointers for E and F to be resident.

3 Analysis

The following section analyzes and provides metrics for the performance of RebFS under three particular use cases.

3.1 Repeatedly Writing to a Small File

As described in section 2.4.3, writing to a small file results in the creation of a new version at most once between the `open()` and `close()` system calls. We analyze the worst case when a write is being performed for the first time and a new version is created. Creating a new inode for a small file and setting all resident bits to 0 takes a constant amount of work. For every block that is written, we must update the corresponding block pointer in the previous version to be resident. This method takes time in proportion to the number of blocks being written. In terms of space, each new version requires an additional inode and a number of blocks equal to the number of blocks being written.

3.2 Repeatedly Writing a Block of a Large File

RebFS excels when repeatedly writing a block to a large file. We assume that the file is large enough that it has data referenced by double or triply indirect blocks. Writing to the beginning of the file results in identical performance as writing to a small file. When creating a new version of the file, the resident bits in all block pointers are set to 0. No indirect block pointers need to be followed. Writing to a block later in the file requires only a small amount of additional work. For any block not directly referenced by the inode, before setting its pointer to resident, its parent

block must also be resident. A versioned file is not permitted to modify a block it does not own. Thus, writing blocks referenced triply indirectly requires space equal to the number of blocks being written plus three additional blocks.

Alternative designs using other types of block reference counting would be inherently slower when handling large files. For example, keeping track of counts in the block bitmap would require every indirect block to be expanded to update reference counts during garbage collection as well as automatic versioning. This overhead would result in poor performance when repeatedly writing blocks to a large file.

3.3 Searching All Versions of a Small File

Searching through all versions of a small file is similar in performance to searching for a particular string in all files in a directory. Each version in a version set references its blocks in the same way as regular inodes. To optimize the search, we hash the results of searching in each block by the corresponding block pointer. Since each block may be referenced more than once, we lookup the result of the search in the hash table before following the block pointer and performing the search. This method makes searching for an occurrence of a string across all versions of a file higher in performance than looking for the string in the same number of individual files of the similar length.

4 Conclusion

RebFS is a high-performance versioning file system based off of the UNIX file system that stores versioning information into data structures called version sets. By reusing blocks that versions of a file have in common, it stores many versions of a file in a space-efficient manner. Automatic garbage collection and customization of which files are versioned allow for a high degree of user control. Further investigation regarding optimizing the storage of the blocks of a version in a sequential manner might improve the overall performance of the system. In addition, a defragging algorithm should be designed to work with version sets. A viable testing plan must be in place to ensure the implementation of this versioning file system is a successful one.

5 References

[1] K. Muniswamy-Reddy. (2004, April). *A Versatile and User-Oriented Versioning File System* [Online]. Available: http://www.eecs.harvard.edu/~kiran/pubs/versionfs-fast04.pdf

Word Count: 2498