

Carbon Copy Filesystem: A Real Time Snapshot Filesystem Layer

David Letscher
Dept. Mathematics and Computer Science
Saint Louis University
221 North Grand Blvd
St. Louis, Missouri 63103-2007
letscher@slu.edu

January 16, 2006

Abstract

Snapshotting is a common strategy for backing up filesystems at regular intervals. We describe a mechanism for integrating the snapshots into an existing filesystem providing straightforward users direct access to backups of their files. The filesystem is implemented in user space using FUSE and wraps an existing filesystem providing the backup framework.

1 Introduction

System users often make changes to or delete files and discover it was a mistake. If the user is fortunate, a backup of the previous version of the file may exist. Often, retrieving the backup version requires the involvement of the system administrator or use of specialized utilities. A user can also ensure that they have access to previous versions by doing their own backups or using a version control system [1, 11]. In any of these options either the user must learn how to perform backups, use version control or seek assistance. We propose a filesystem layer to automate the backups through snapshots and provide straightforward user access to backups.

Snapshots and other forms of automated backups of a filesystem can be done in a variety of ways. The lowest level of these would be taking the backups at the block level. Both LVM and EVMS have mecha-

nisms for this [8, 4]. However, there access to these backups is not straightforward and require administrator level access.

Some filesystems, including AFS and Ext3cow have built-in snapshot capabilities [9, 6]. Backup recovery for many of these filesystems either require administrator access or other special utilities. Other filesystems, including Cedar [5] and Elephant [10], integrate version control; backing up every version of a file. Versioning has also been done using a layered filesystem built upon existing filesystems with versionfs [7] and wayback [3] With complete versioning there is additional overhead in disk space and access times and may not be ideal for longer term backup solutions on some systems.

Another option is to use various utilities running at predetermined times to create the snapshot. Many of these utilities are written in scripting languages and often make use of rsync. These systems can be used on top of existing filesystems and are useful for a variety of purposes. However, they do not provide seamless access to the user. They also have the disadvantage of using a large amount of system resources when the snapshot is created. This may cause problems and would not work well when snapshots are to be taken often.

1.1 Design Goals

Considering existing systems for providing backups we have the following goals for the CarbonCopy filesystem:

Transparency The user should be able to access saved versions of files using existing filesystem utilities with no involvement of the system administrator or use of specialized utilities.

Low overhead Performance hit should be minimal when dealing with current versions of files. Additional overhead for accessing older versions of files is acceptable.

Disk usage Additional disk usage should be minimized, when possible. It should be possible to store the snapshots on a different device to reduce the impact of a drive failure.

Versatility Should be able to work with many existing filesystems.

Straight forward Implementations should be done with minimal amount of code.

Single point in time The snapshots should image the filesystem as it was at a single point of time.

2 User's Perspective

One of the goals for CCfs was to make its usage seamless from the users perspective. In particular, the user should be able to access all files using existing shell utilities, file browsers, etc. Accessing current versions of files is identical to ordinary file system access. And accessing previous versions is done through simulated directory structures and symbolic links.

In figure 1, the state of an example filesystem at two different points in time. We will assume that a snapshot has occurred between these two times and that the file `sources/code.cc` has been modified after the snapshot was taken. The current versions of the files appear in the directory tree appear as usual.

Access to previous versions is through hidden "virtual" directories, see figure 2. When the user does an `ls -a` or says to show hidden files in a file browser,

they will see a subdirectory `.SNAPSHOT` in every one of their subdirectories. These "virtual" directories contain directories for the available timestamps that snapshots were taken. All of the backed up files cannot be written to, regardless of the file permissions, and cannot be removed by the user.

3 System Administrator's Perspective

The filesystem compiles as a kernel module that once loaded allows mounting to occur as any other filesystem and a small collection of utility software. Various parameters, such as how often snapshots should be taken, are controlled through the `/proc` filesystem or through mount options. To mount a carbon copy filesystem, a base filesystem must already be mount for it to store its files on. This filesystem can be any block based filesystem; to date it has been tested with ReiserFS, NFS, Ext3, JFS and XFS. This base filesystem will still be directly accessible; however, direct access is not recommended as it could cause problems with the snapshots. If desired, a second existing filesystem can be used to store the snapshots providing some level of redundancy. This can be used to ensure that a drive failure cannot wipe out both the current file versions and the backup copies. A typical mount command would be:

```
mount -t ccfs none /ccfs -o base=/basefs
```

And if a separate filesystem is used to store the snapshots:

```
mount -t ccfs none /ccfs -o \\  
base=/basefs,backup=/backupfs
```

Control parameters, such as how often timestamps should be taken, policies for removing old versions and compression level can be found in `/proc/ccfs`. These options can also be set with further mount options.

The filesystem includes several administrative utilities:

ccfscreate Create a new Carbon Copy filesystem from an existing filesystem.

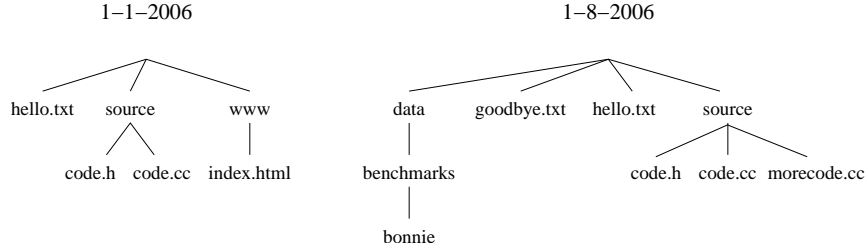


Figure 1: Filesystem view at two different dates.

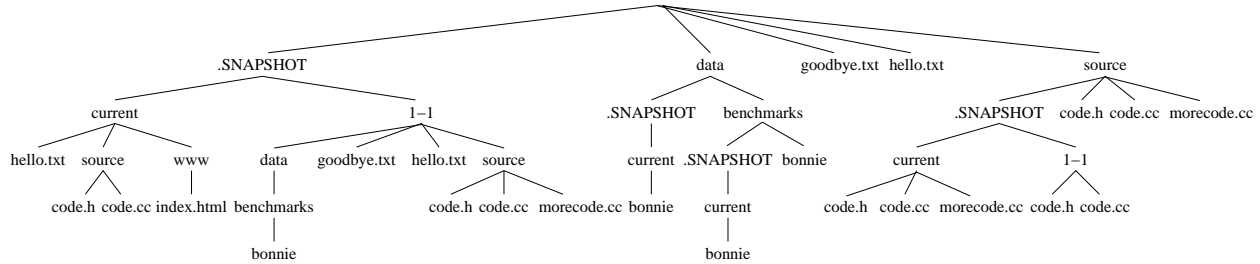


Figure 2: Users' view of same filesystem in CCfs.

ccfsrm Used to remove old snapshots.

ccfsopt Optimize the filesystem. This is usually used to reduce disk space by compressing old file versions.

ccfsfind Find all of the different versions of files.

4 Implementation

4.1 FUSE

Creating a new filesystem from scratch or modifying an existing filesystem for snapshotting, as was done with EXT3COW [9], can be difficult and time consuming. A more straight forward approach is to write a filesystem layer to put over an existing filesystem. One approach to do this is the use FiST templates [13] which are specifically designed for such a task. Another option is to write a kernel module using the Linux virtual filesystem interface. A third option is to implement the filesystem in user space

using FUSE [12]. In addition to being more straightforward, the option of using FUSE also allows easy integration of existing userspace utilities, such as gzip and diff.

FUSE has language bindings for several languages including Python [12]. This allowed rapid development of a prototype system. A full implementation of the layered filesystem was written in approximately two hundred lines of code. This test version turned out to be fairly quick and provided a great test base to iron out the details of the implementation. After the finalization of the design, the filesystem was rewritten in C using the FUSE library.

4.2 Redirection

The CCFS layer of the filesystem primarily redirects any filesystem to the appropriate file on the base filesystem. Figure 3 shows how the files are stored on the base filesystem is stored for the example talked about in the previous section. A file stored on the base filesystem has is named in the format

FILENAME_ccfs_START_END where FILENAME is the actual name of the file, and START and END are the range of timestamps that this is the version of FILENAME to use. Note these timestamps refer to times that snapshot were taken, not when a file was create or modified. Depending of compression level these filenames may have extensions of .gz or .diff.

Reading From the Current Version When a file is open for reading than a filename of the form FILENAME_ccfs_*.current is search for. If no such file exists an error occurs, otherwise it is opened for reading and its file pointer is return. All operations on this file are passed on to the base filesystem.

Reading From a Previous Version When access is requested to a .SNAPSHOT subdirectory the timestamp desired for a file can be determined by which subdirectory is given. A particular file may remain unchanged for a range of timestamps. A filename of the form FILENAME_ccfs_*. is search for that contains the desired timestamp. This file can then be opened readonly and all file operation requests can be passed on to the base filesystem.

Writing or Re-writing Opening a file for writing or rewriting can be a little more complicated. If the file does not exist, a file with the appropriate name is created on the base filesystem. However, if the file already exists then one of two things occur. If the file is not part of a snapshot, so its name is of the form FILENAME_ccfs_current_current, then all file operations can be redirected to this file. If the file should be part of a snapshot, i.e. its name is of the form FILENAME_ccfs_TIMESTAMP_current, then we must duplicated the file. The two versions will be FILENAME_ccfs_TIMESTAMP_LASTSNAPSHOT and FILENAME_ccfs_current_current, where LASTSNAPSHOT is the time when that last snapshot was taken. All file operations are then redirection to the current version of the file.

Deleting a File When a file is to be deleted there are two possibilities. If the current version of the file has changed since the last snap-

shot then it can be deleted. If it has remained unchanged since that snapshot, it is renamed from FILENAME_ccfs_TIMESTAMP_current to FILENAME_ccfs_TIMESTAMP_LASTSNAPSHOT.

Compression If compression is turned on, then a background task is used to do the compression of snapshot files. There are four compression levels:

none No compression is performed on snapshotted files.

gzip All snapshot files are stored in gzip format

diff All snapshot files are stored as binary diffs relative to the previous snapshot. The most recent snapshot is stored unchanged.

diff+gzip Diffs are kept of snapshots as aboveand then compressed with gzip. Most recent snapshots are also compressed.

This compression starts in the background whenever a file is no longer current and is triggered when a file is opened for writing or deleted. Since this occurs online there is no extra work to be done at the point in time a snapshot is to be taken.

Clearing Out Old Versions One of the settings for the filesystem is a policy for how long to keep old versions. However, if there is room on the device, older version are kept longer and not deleted until the space is needed. Once freespace falls below a set threshold, old versions are deleted on the base filesystem as more room is requested. These files are only removed as necessary and only files older that the policy says to keep are deleted. The storage used for these files that no longer needs to be kept always count as part of the free space on the device.

5 Performance

5.1 Bonnie Benchmark

The Bonnie benchmarks measure raw disk read and write operations using large files [2]. They provide a good measure for determining once a file is opened

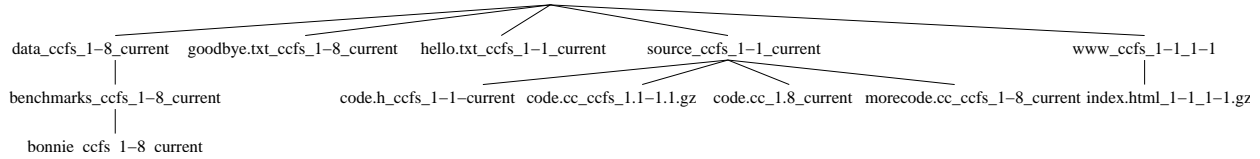


Figure 3: Underlying representation of CCFs filesystem on base filesystem.

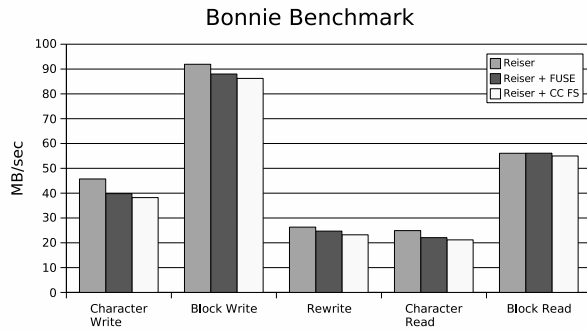


Figure 4: Read/write speeds on Bonnie benchmark.

how fast can it be written to or read from. Our test system is an AMD Athlon X2 3800 with 2 GB RAM and a 250 GB Western Digital SATA drive with 16 MB cache.

All of these tests we using newly created files, so access to previous snapshots never occurs. So these tests are not relevant to read/write speeds to earlier versions.

The results of these tests are in figure 4. These tests were done with a base filesystem of Reiser 3, a base FUSE filesystem that passed all requests to the base filesystem and the CarbonCopy filesystem built on top of the other two. This breakdown was done to distinguish overhead due to the layering of the filesystems and the operations of CCFS.

The chart shows the throughput in MB/sec for each test. The CCFS throughput is between 84% and 98% of the base Reiser filesystem with the biggest impact of single character read and writes. The impact on block read and writes is between 94% and 98%. On this system, the overhead for basic file operations

is relatively small.

5.2 Other Benchmarks

Another set of benchmarks run is a modified version of the benchmarks for the Andrew filesystem [6]. These tests used the source tree for the the 2.6.15 Linux kernel. Five tests were performed:

untar Building the source tree for the Linux 2.6.15 kernel from the `.tar` archive. This focuses on testing the creating of new directories and files and writing to these files.

stat Perform an `ls -l` for all files in the source tree. This tests the listing of files and access to their metadata.

wc Do a wordcount on all source files. This primarily measures sequence access to the entire contents of the file.

make Build the kernel. This tests both the reading of files and writing of files with the system under more stress.

make clean Remove object files from the directory tree. This measures the removal of files from the filesystem.

Each of these tests were done on the base Reiser filesystem with times ranging from a third of a second to four minutes. Because of this wide range of times, all times were measured relative to the time it took for the base filesystem. As with the previous tests, they were also done on the FUSE passthrough filesystem to measure the overhead of layering the filesystem.

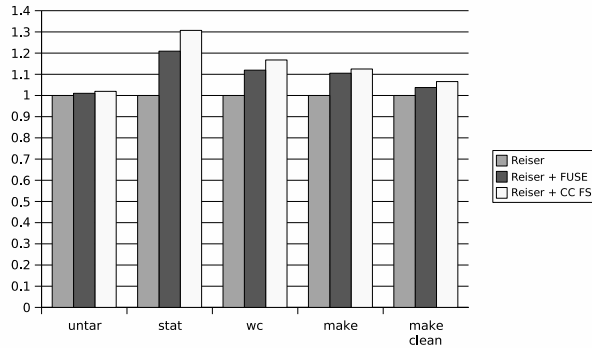


Figure 5: Performance on a clean filesystem as multiples of times for Reiserfs.

The first round of tests were done on a clean filesystem on the same setup as the Bonnie tests were performed. The results are shown in figure 5. The metric is the multiple of the time the given test took on the base Reiser filesystem.

The times to perform these various operations ranged from approximately the same time as with Reiser up to a slowdown of about 31%. The largest slowdown was then access statistics about all of the files. The data suggests that the biggest slowdown occurs in the path redirection and finding the current version of the file.

A second round of tests were performed when there was an copy of a previous version of the Linux kernel in the filesystem. In this case the previous version was 2.6.14 and it is assumed that this previous version has been snapshotted by CCFS. These tests were done for CCFS without any compression, gzip'ing old versions, using binary diffs and using both file diffs and gzip compression.

The results of this round of testing is in figure 6. As with the previous tests, the metric is multiples of the time it took for the tests to run on the base Reiser filesystem. The only significant slowdown was creating new files with further slowdowns due to compression. Note that this only affects access times the first time the file is written to after a snapshot. Subsequent accesses have minimal slowdown.

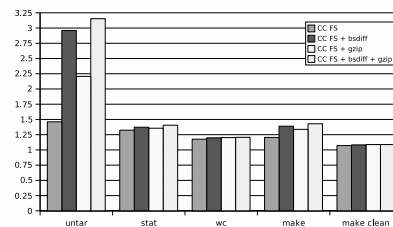


Figure 6: Performance after previous version of the kernel has been snapshotted as multiples of times for Reiserfs.

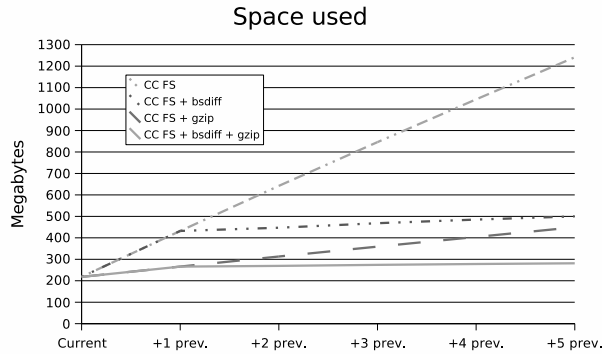


Figure 7: Time to access current and previous versions of the Linux kernel.

5.3 Access to Snapshot Versions

The previous versions of files are stored in a read only format that may or may not be compressed. The next set of tests measure access to these versions. The setup uses kernel versions 2.6.10 thru 2.6.15, with a snapshot taken after each version. Each version has much in common with previous versions but there are still many differences.

To test the read time, a word count of each file in the source tree in performed the results are shown in figure 7. Access to the current version is approximately the same as we saw in the previous tests. Without compression, access times stay consistent from version to version. And if no diffs are taken, access times do not grow as older versions are accessed. However, access times increase roughly linearly when diffs are used. This is due to the fact that diffs are relative to the previous version. So more calculations are necessary to rebuild these older versions.

5.4 Disk Usage

Finally the disk usage was measured for each compression option based on the number of snapshots kept. This was also done for the Linux kernel versions 2.6.10 thru 2.6.15. The results are in figure 8. Without compression the disk space requirements were approximately linear in the number of versions kept.

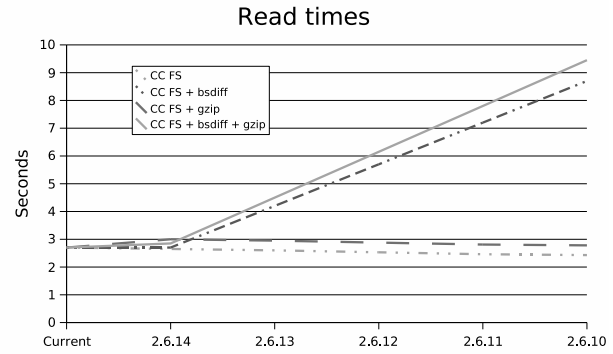


Figure 8: Space used for storing the current Linux kernel and several previous versions as snapshots.

With any forms of compression disk usage grew far slower. Storing all six kernel versions only took approximately 50% additional space at the maximum compression level.

6 Conclusions and Future Work

The Carbon Copy filesystem has met all of the design goals. It has turned out to be easy to use for the average user and makes dealing with backups much easier for the system administrator. As seen in the previous section, the performance was nearly as good as the base filesystem. For many situations this extra overhead is a justifiable price for ensuring easily accessible backups exist. Working as a layered filesystem over virtually any existing filesystem makes it easy to install and administer.

This version of cfs was done in approximately 1800 lines of code. This was made possible by using FUSE along with userspace utilities for compression and binary diffs. Compared to the effort of writing a new filesystem or modifying an existing filesystem, this was very straightforward and easier to implement.

The main place for improvements is the reduction of overhead of the system. From a design point of view, this could be accomplished by not using FUSE and instead accessing the virtual filesystem directly.

This could increase the speed of the system at the cost of a more complex design and significantly larger codebase.

References

- [1] B. Berliner and J. Polk. Concurrent versioning system (cvs). <http://www.cvshome.org>.
- [2] Bonnie. <http://www.textuality.com/bonnie>.
- [3] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A user-level versioning file system for linux. *Proceeding of Usenix*, 2004.
- [4] Enterprise volume management system. <http://evms.sourceforge.net>.
- [5] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The cedar file system. *Communication of the ACM*, 31(3):288–298, 1988.
- [6] J. H. Howard and et al. Scale and performance in a distributed file system. *Transactions of Computer Systems*, 1988, February 1988.
- [7] A. H. Kiran-Kumar Muniswamy-Reddy, Charles P. Wright and E. Zadok. A versatile and user-oriented versioning file system. *Proceeding of the 3rd USENIX Conference on File Storage and Technologies*, 2004.
- [8] Logical volume manager. <http://www.tldp.org/HOWTO/LVM-HOWTO/>.
- [9] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.
- [10] D. S. Santry and et al. Deciding when to forget in the elephant file system. *Proc. of the 17th ACM Symposium on Operating System Principles*, 1999.
- [11] Subversion. <http://www.subversion.org>.
- [12] M. Szeredi. Filesystem in USEr space. <http://sourceforge.net/projects/avf>.
- [13] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of USENIX*, 2001.