Wayback: A User-level Versioning File System for Linux

Brian Cornell Peter A. Dinda Fabián E. Bustamante
Computer Science Department, Northwestern University
{techie,pdinda,fabianb}@northwestern.edu

Abstract

In a typical file system, only the current version of a file (or directory) is available. In Wayback, a user can also access any previous version, all the way back to the file's creation time. Versioning is done automatically at the write level: each write to the file creates a new version. Wayback implements versioning using an undo log structure, exploiting the massive space available on modern disks to provide its very useful functionality. Wayback is a user-level file system built on the FUSE framework that relies on an underlying file system for access to the disk. In addition to simplifying Wayback, this also allows it to extend any existing file system with versioning: after being mounted, the file system can be mounted a second time with versioning. We describe the implementation of Wayback, and evaluate its performance using several benchmarks.

1 Introduction

A user of a modern operating system such as Linux experiences a very simple file system model. In particular, the file system only provides access to the current versions of his or her files and directories. The trouble with this model is that the user's progress in his work is not monotonic – the user makes mistakes such as discarding important text in a document, damaging carefully tuned source code through misunderstanding, or even accidentally deleting files and directories. Beyond mistakes, it is often helpful, especially in writing code or papers to look at the history of changes to a file. If the user could "go back in time" (using the "wayback machine" from Ted Keys's classic cartoon series "Peabody's Improbable History," for example), she would be in a better position to recover from such mistakes or understand how a file got to its current state.

Of course, version control systems such as RCS [RCS] and CVS [CVS] provide such functionality. However, the user must first become familiar with these systems and explicitly manage her files with them. In particular, the user must tell these systems when a new version of the files is ready by explicitly committing them. Hence, the user determines the granularity of versions, and, since he must explicitly make them, they tend to be large. Some tools such as EMACS include drivers to automate this process. Some applications (e.g. Microsoft Word) provide their own internal versioned file format. Here the versioning is usually done automatically at the granularity of a whole editing session.

We believe that a better way to help the user revert to earlier versions of her file is to automatically provide versioning in the file system itself, and to provide it at a fine granularity. To this end, we have developed the Wayback versioning file system for Linux. With no user interaction, Wayback records each write made to each file or directory into a permanent undo log [UNDO]. The undo log can then be unwound in reverse order (prompted by a user-level tool) to rollback to or extract any previous version of a file. Wayback is implemented as a user-level file system using the FUSE kernel module [FUSE]. It operates on top of an existing, non-versioned file system, adding versioning functionality.

Versioning file systems have been around for quite some time. It was already provided by some early file systems such as Cedar File System [CEDAR] and 3-DFS [3DFS]. The VMS operating system from DEC introduced versioning to a broad range of users. The VMS file system created a new version of a file on each close [VMS]. Checkpointing is an alternative approach to provide versioning: snapshots of the entire file system are taken periodically and made available to the user. Example of systems using checkpointing include AFS [AFS], Petal [Petal] and Ext3Cow [Ext3-Cow]. One limitation of checkpoint-based versioning is that changes made between checkpoints cannot be undone. The Elephant versioning file system was among the first to recognize that versioning was a excellent way to exploit the massive (and exponentially growing) disk sizes that are available today [Elephant]. There has since been an explosion of interest in versioning file systems. Wayback does versioning at the level of writes and hence is a *comprehensive* versioning file system [CVFS]. Its ability to run on top of an existing file system is similar to the concept of a stackable file system such as Versionfs [VersionFS2]. Versionfs implements some of the same functionality of Wayback on Linux [VersionFS1], but there is no public release available. As far as we are aware, Wayback is the first public release (under the GPL) of a versioning file system for Linux.

2 User's View Of The Wayback FS

Wayback FS requires a recent 2.4 or 2.6 Linux kernel, gcc 2.95.2 or higher, and Perl 5. We have used kernel versions as early as 2.4.7-10 (shipped with Red Hat 7.2). The FUSE user-level file system module is used (versions 0.95 and up). The current Wayback distribution is shipped along with the FUSE source. Compilation involves the typical make, make install routine. The output includes:

- fuse.o: FUSE kernel module
- wayback: Wayback FS user-space server
- vutils.pl: command-line utility for manipulating files.
- mount.wayback: easy mounting script

Four symbolic links to vutils.pl are also created to expose its basic functions:

- vstat: Describe a versioned file.
- vrevert: Revert a versioned file to an earlier version.
- vextract: Extract a specific version of a file.
- vrm: Permanently remove a file.

To mount a Wayback file system, the underlying file system is first mounted in the normal manner, then it is remounted by starting a Wayback server:

$ wayback path-in-underlying-fs mount-path

A script named mount.wayback is included in the distribution that remounts paths nicely such that all users can access the versioned files as they could the underlying files. mount.wayback is executed with the same options as wayback above.

After this, the user can access his files through mount-path. Any change made will be logged and is reversible using vrevert. Old versions can also be copied out using vextract. Even "rm" is logged and can be undone. To permanently remove a file, vrm is used. Notice that it is possible to mount the directory hierarchy under any path as a new, versioned, file system. It is also possible to continue to manipulate files in the original path, but those changes will not be logged and are not revertible.
Versions are tagged in two ways: by change number (starting with one being the most recent change) and by time stamp. The user most often uses the time stamp, it being natural to revert or extract a file or directory as it existed at a particular point in time.

3 Implementation

Wayback FS is implemented as a user-space server that is called by the FUSE kernel module. In essence, FUSE traps system calls and upcalls to the Wayback server. The server writes an entry into the undo log for the file or directory that reflects how to revert the effects of the call, and then executes the call on the underlying file system. We opted for FUSE because of familiarity with the tool. We could have alternatively employed SFS [FiST].

3.1 Log Structure For Files

Each file for which versioning information exists has a shadow undo log file, named by default "<filename>. versionfs! version". Each log record consists of:

- A time stamp,
- The *offset* at which data is being overwritten or truncated,
- The *size* of the data that is being lost,
- Whether the file size is increasing due to writing off the end of the file, and
- The *data* being lost, if any.

3.2 Logging File Operations

Wayback traps calls to write() and truncate() for files. Every time write() is called on a file, versionfs reads the portion of the file that the write call will overwrite and records it in the log. The *offset* recorded is the offset in the file at which data is being written, the *size* is either the number of bytes to the end of the file or the number of bytes being written (whichever is smaller), and the data is the *size* bytes at *offset* that are currently in the file.

When truncate() is called on a file, the *offset* recorded is the length to which the file is truncated, *size* is the number of bytes in the file after that point, and *data* is that data that is being discarded due to truncation.

3.3 Log Structure For Directories

Every directory has a shadow undo log that we call the directory catalog. The directory catalog logs when any entry in the directory is created, removed, renamed, or has its attributes change. The directory catalog has the default name “<directory>/. versionfs! version”. Each log record consists of:

- A time stamp,
- The *operation* being performed,
- The *size* of the data recorded for this operation, and
- The *data* needed to undo the operation. The interpretation of the data depends on the operation.

3.4 Logging Directory Operations

When mknod(), mkdir(), or creat() is called, a link is created, or open() is called with the O_CREAT flag, the directory catalog is updated with the create or mkdir operation number and *data* consisting of the filename that is being created.

When unlink() is called on a regular file, the file is first truncated to zero length to preserve the contents of the file before deletion. Next, the directory catalog is updated with *data* consisting of the attributes of the file (mode, owner, and times) and the filename that is being deleted. For links, the destination is also recorded.

Calls to rmdir() in Wayback actually translate to calls to rename(). Directories are *never* deleted because their contents would be lost. Instead an identifier such as “. versionfs! deleted” is added to the directory name. Subsequently, and for user-initiated rename() calls, the directory catalog is updated with *data* consisting of the old name of the file or directory, and the new name of the file or directory.

When chmod(), chown(), or utime() is called, the directory catalog is updated with *data* consisting of the attributes of the file and the filename for which attributes are being changed.

4 Design Issues

We encountered several issues while designing and implementing Wayback. The solutions we found and decisions we made have defined what Wayback is now.

4.1 Kernel Versus User-level

The first major decision we had to make was whether this file system should be implemented in the kernel as its own file system, or using a user-level module. The trade-offs are in speed, ease of implementation, and features. A kernel module would undoubtedly be much faster because the user-level overhead would be avoided. However, it could limit compatibility to certain kernel versions, and it would preclude adding versioning to existing file systems. It would also be much harder to implement a kernel module.

The main factor in our decision to make a user-level file system had to do with the features we could easily implement. We considered writing Wayback as a kernel-level extension to ext3. This would probably have been faster, but it would have been limited to ext3 file systems on normal block devices. Implementing Wayback as a user-level file system would make it slower, but would let us remount any file system with versioning.

4.2 Choice of Undo Logging

Wayback logs changes as undo records. We recover previous versions by applying these records in reverse order until the appropriate version is reached. This is straightforward, but it has a downside: while reverting to newer versions is very fast, reverting to very old versions can take some time. One alternative is a redo log, in which modifications themselves are written as log records. Recovering an old version means applying the records in forward order until the appropriate version is reached. This has the advantage of allowing very old versions to be recovered very quickly, but newer versions are slow. A third possibility is an undo/redo log, which contains both undo and redo records, allowing us to move backward and forward easily. Each logging technique can be combined with periodic checkpointing, providing snapshots of the whole file state from which to move forward or backward using the log.

We chose simple undo logging for Wayback because we felt that for our use cases – reverting mistakes made in editing programs and documents – we would typically have only to move backward by a small number of versions. In light of other applications we would like to support (see Conclusion), we are reconsidering our logging model.

4.3 Use of FUSE

Once we had decided on a user-level approach, we next considered how to interact with the kernel. At the time we started development, FUSE was still in its early stages (we started with FUSE 0.95), but being able to avoid kernel development altogether was very tempting to us since we wanted to concentrate on the versioning mechanisms. The FUSE kernel module provided us with the level of access we needed on a modern Linux kernel. FUSE proved to be relatively stable and easy to use.

Early versions of FUSE did not provide an upcall for the close() system call. This lack would have made it impossible to create new versions on close, as in VMS. Fortunately, we had determined to do write-level versioning. However, it still indirectly affected Wayback's design. In particular, without close() calls, managing file descriptors for log files is made unnecessarily difficult.

4.4 Path Redirection

After deciding to use FUSE, we quickly came upon another issue. FUSE is designed only to provide a destination path for the file system, and not a source path to mount there. The examples for FUSE either remount an entire directory structure beginning in the root directory, or provide their own root from a different source such as memory.

We decided that we wanted to have redirection from any path, not just the root directory, so we had to implement a work-around. Wayback takes different command-line arguments from other FUSE file systems, and then modifies those arguments before passing them on to FUSE. We then use the information from those arguments to modify every path given to Wayback from FUSE, redirecting it to the "real" path.

5 Performance Evaluation

A variety of performance tests were run on Wayback FS to evaluate its performance. These tests include Bonnie [Bonnie], which performs various operations on a very large file; the Andrew Benchmark [Andrew], which simulates the operation of normal file systems; and a test that compares using Wayback FS to using manual checkins with RCS.

These tests were all run on three test systems:

- Machine A: AMD Athalon XP 2400+ with 512 MB of RAM using an internal 2.5 inch notebook hard drive.
- Machine B: Intel Pentium 4 2.2 GHz with 512 MB of RAM using an external USB 1.1 disk (1.5 MBps).
- Machine C: Intel Celeron 500 MHz with 128 MB of RAM using an internal 2.5 inch notebook hard drive.

All of the tests were run under Linux kernel 2.4.20 or 2.4.22.

For comparison, another file system was built on top of FUSE before tests were run. This file system simply redirects requests from the mounted file system through to the underlying file system, acting as a pass-through. This file system is used to identify the performance hit taken solely by the FUSE system, and isolate the performance loss from versioning. We consider the following file systems:

- ext3: the out-of-box ext3 file system
- ext3+fuse: ext3 run through our pass-through FUSE file system
- ext3+ver: Wayback FS running on top of ext3

These configurations were used for our Bonnie and Andrew tests. In comparing to RCS, we compared "ext3+ver" to "ext3", where the files in question were periodically committed using RCS.

We did not run any tests on the performance of reverting files, because it is not an everyday occurrence and shouldn't matter as much as reading and writing performance. Reverting or extracting a recent state from a file typically takes at the most seconds if not less than a second. Disk space usage does increase when reverting files depending on the size of the reversion, because Wayback does not remove the reverted entries from the log. Rather it runs them backwards on the file, creating more entries in the log.

5.1 Bonnie

Bonnie was originally created by Tim Bray to identify bottlenecks in file systems. It does so by creating a very large file, large enough to require disk access rather than caching, and reading from/writing to that file as fast as it can.

5.1.1 Bonnie Implementation

Bonnie is implemented as a single C program. The program takes arguments for the size of the file it will use and the path where it will be stored. Bonnie then creates the large file and performs many sequential operations on it, including writing character by character, writing block by block, rewriting, reading character by character, reading block by block, and seeking. For this test, Bonnie was run with a file size of 1 GB.

5.1.2 Bonnie Results

Figures 1-3 show the performance of the different Bonnie phases on the three machines. For each phase and machine, we compare ext3, ext3 via our pass-through FUSE file system, and ext3 with Wayback versioning. The performance metric is in KB/s as measured by Bonnie. The point is to measure how much performance is lost by using Wayback and how it breaks down into FUSE overheads and the actual costs of Wayback. Figure 4 shows the CPU costs, in terms of percentage of CPU used as measured by Bonnie.

It is important to point out that in some cases layering ext3 below FUSE actually increases its performance. We expect that this is due to buffering effects as there is now an additional process which can buffer. Additionally, the overheads shown in Figure 4 are slightly misleading. Bonnie is measuring the system and user

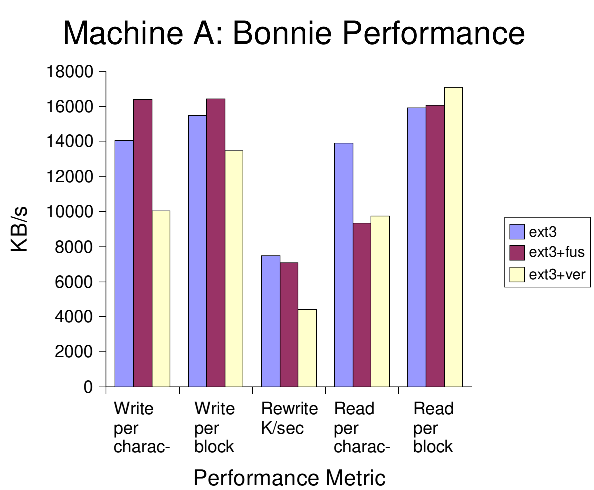

Figure 1. Bonnie Performance on Machine A

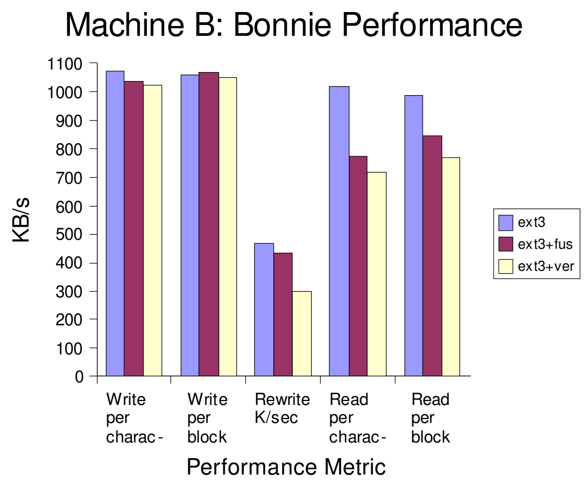

Figure 2. Bonnie Performance on Machine B

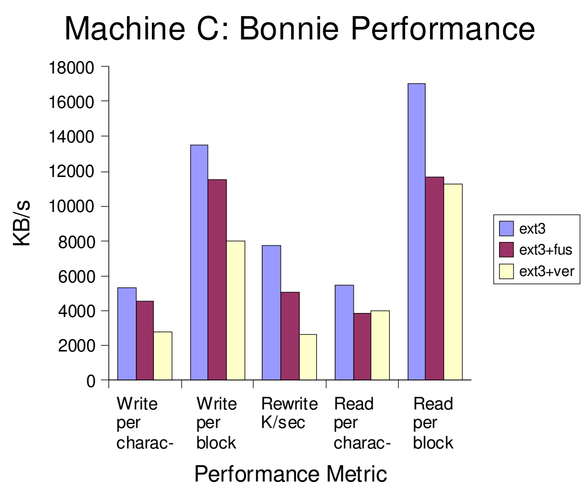

Figure 3. Bonnie Performance on Machine C

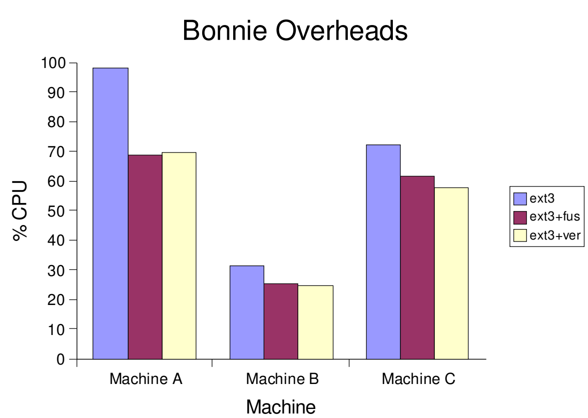

Figure 4. Bonnie Overheads

time it uses, but does not count the time spent in the Wayback server on its behalf.

For block writes in Wayback, we see performance impacts in the range of -2% to -40% compared to unadorned ext3, depending on the speed of the disk and the machine. For block reads, the performance impact is +5% to -32%. Character writes are impacted -3% to -50%, while the character read impact is -10% to -30%. In re-writing, where would expect to see the maximum impact, the range is -30% to -70%.

The largest impact on write speed is on machines with fast disks, particularly those that also have slow processors. The largest impact on rewrite speed is on machines with slow disks, which is to be expected as rewrites will include additional data to be written to the logs. Read speed is maximally affected on slow machines with fast disks. In many cases, a large portion, often the majority of the performance impact is due to FUSE rather than versioning.

5.2 Andrew Benchmark

The Andrew Benchmark, although quite old, is commonly used to measure the performance of file systems. It performs operations similar to those performed every day on file systems, including making directories, copying files, stating and reading files, and compiling a program.

5.2.1 Andrew Implementation

The original Andrew Benchmark was written on and designed for Sun computers. It consists of a makefile that executes operations in five phases, ending in the compilation of a program. The program used in the benchmark will only compile on Sun systems however. The Andrew benchmark also only runs each phase once, and does not delete the files it creates.

Because of these limitations, we rewrote the Andrew Benchmark in Perl. The program runs the same phases as the original Andrew Benchmark, except that it can run them with any set of files. It can also run the test multiple times and print a summary.

The phases of the Andrew Benchmark are designed to emulate everyday use of a file system. The phases are all done using the source directory of a program, and include:

- Phase 1: Create five copies of the directory structure in the source tree.
- Phase 2: Copy all of the files from the source tree into the first set of directories created.
- Phase 3: Stats each file using `ls -l`.
- Phase 4: Read each file using grep and wc.
- Phase 5: Compile the source in the test tree.

The source that we used is that of the window manager ION. Each phase was executed 1000 times to get accurate results. As before, we ran the benchmark three times on each time, once with the ext3 file system, once with the pass-through file system on ext3, and once with Wayback FS on ext3.

5.2.2 Andrew Results

Figures 5-7 compare the performance of the different file systems for each phase on each machine. The performance metric is the average wall-clock time to run each phase. Phase 5 (Compilation) times have been divided by 20 to fit on the graphs.

There are several takeaway points from these graphs. First, the largest performance impact of Wayback is on directory creation (Phase 1). Second, Wayback increases the time to run the write-intensive copy phase (Phase 2) by between a factor of two and a factor of four. The largest impact is, not surprisingly, on a machine with a slow disk. Wayback has negligible impact on the stat phase (Phase 3), except on very slow machines. The impact on reads (Phase 4) is relatively low (30%) regardless of the machine or disk. Finally,

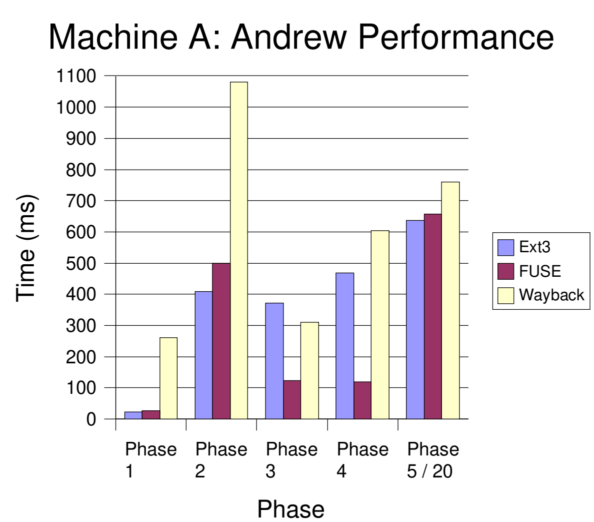

Figure 5. Andrew Performance on Machine A

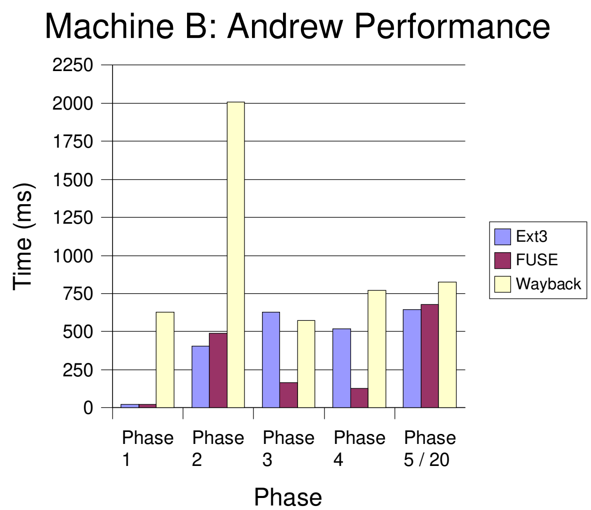

Figure 6. Andrew Performance on Machine B

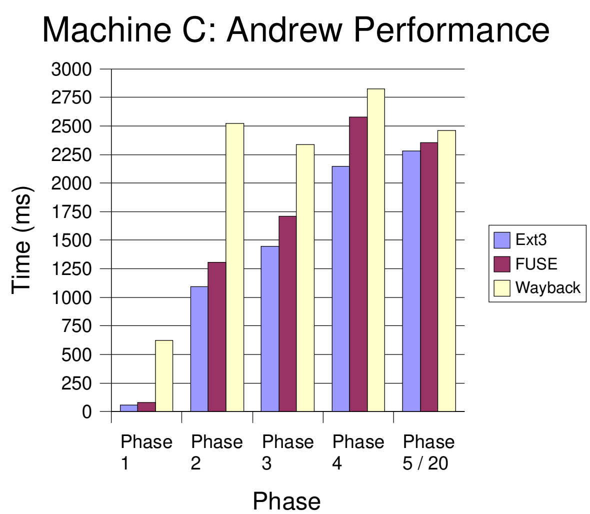

Figure 7. Andrew Performance on Machine C

for compilation (Phase 5), the impact of Wayback is very small (15%) on all three machines.

That the performance impact on compilation is marginal suggests that Wayback could be used very effectively in the edit-compile-debug loop of program development or document preparation with tools such as LaTeX.

5.3 RCS Comparison

In order to test the effectiveness of a versioning file system, it is necessary not only to compare it to other file systems, but to compare it to other methods of versioning. For this reason we have constructed a test that compares different operations on Wayback FS with similar operations using RCS on an ext3 file system.

5.3.1 RCS Implementation

The RCS comparison is implemented as a Perl script that runs through a variety of tests multiple times on both an RCS system and Wayback. The test records the time taken in each case as well as the disk space used.

The RCS comparison runs three modes of testing and produces separate output for the three modes:

- Mode 1: Random seeks within a binary file followed by writing a specified amount of random data. This is designed to emulate normal binary file use. We used 1 MB binary files and 1 KB writes.

- Mode 2: Read an entire binary file into memory, change a specific number of randomly chosen locations with a specified amount of random data, then write the file back to disk. This is similar to the operation of some databases. For this test we used 1 MB files, 1 KB writes, and randomly between 5 and 20 writes per iteration.

- Mode 3: Randomly choose a line in a text file, change a specified number of lines randomly using English words, truncate the file and write everything after the point at which it began changing lines. This test uses a dictionary file to construct files. This is designed to emulate text editing, including changing configuration files and writing code. For this test we used files of 2000 lines, 20 words maximum per line, and changed randomly between 1 and 5 lines per iteration.

Each mode in this test was run for 100 iterations with a file, and the whole ensemble was repeated 10 times with different files. Wayback logged every operation as normal. For RCS, we committed the file periodically, varying the period.

5.3.2 RCS Results

Figures 8-10 show our results. As before, each Figure corresponds to a particular machine. Three curves, one for each mode, are included. Times for the third mode have been multiplied by 10 to fit on the graphs. The vertical axis is the time required to run the mode, while the horizontal axis is the test set. The left-most test set, marked “Version” is for Wayback. The remaining test sets are for RCS with varying period. For example, “RCS 1” corresponds to RCS commits done after every operation, which is equivalent to what Wayback is doing, while “RCS 6” corresponds to RCS commits done after every 6th operation. As the period increases, we amortize more and more of the overhead of using RCS and get closer to the performance of Wayback.

It is clearly the case that Wayback performs far better than RCS at comprehensive versioning. An interesting trend is that for slower processors the difference between Wayback and RCS is greater, and for slower disks RCS nearly catches up to Wayback.

The results show that except in the case of a very slow disk, Wayback performs better with single binary writes (Mode 1) even if RCS is used with a period of 10. On an average system, Wayback performs about as well as using RCS every other time when writing the

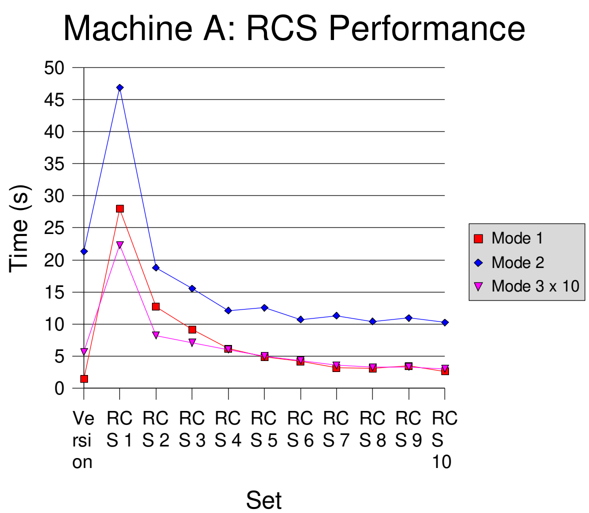

Figure 8. RCS Performance on Machine A

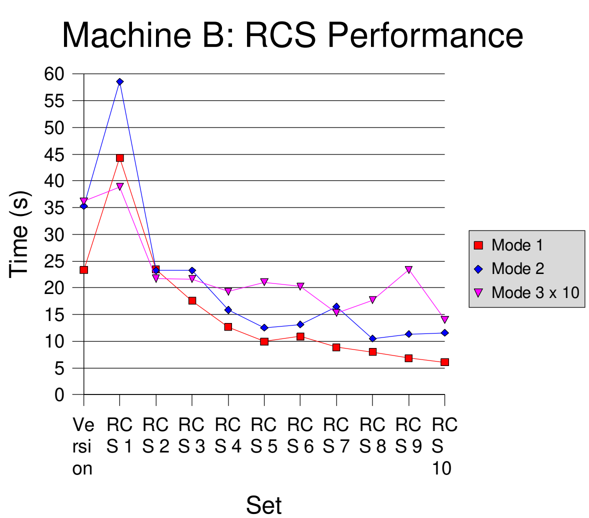

Figure 9. RCS Performance on Machine B

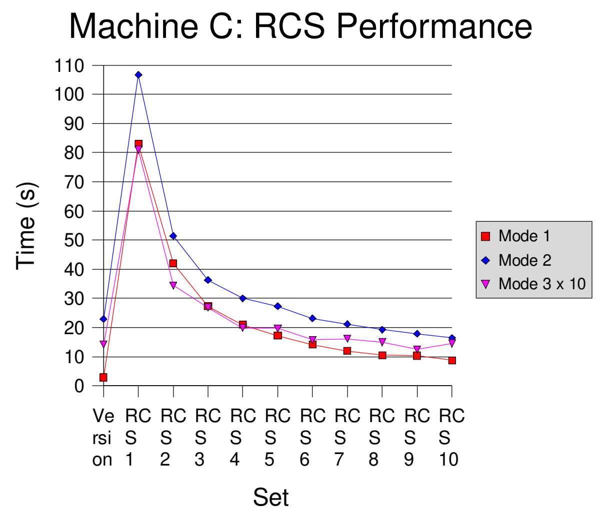

Figure 10. RCS Performance on Machine C

whole binary file (Mode 2). Using Wayback with text on an average system is similar in performance to using RCS about once every four changes.

In terms of disk space use, the results are quite different. For the single binary writes (Mode 1), Wayback uses much less disk space than RCS. For writing the whole binary file (Mode 2), Wayback uses 25 to 30 times as much space as RCS. For text changes (Mode 3) Wayback uses about 20 times as much space as RCS. These results are summarized in Table 1; sizes are shown in bytes and are the average from 10 runs. Disk space is not dependent on the test system, so results are only shown from Machine A.

File Type	Mode 1	Mode 2	Mode 3
Versioned	1157456.4	106347428.0	2182218.0
RCS Period 1	2242325.4	3856521.8	101062.2
RCS Period 2	2237020.7	3779180.4	96134.2
RCS Period 3	2233854.1	3731427.8	94336.4
RCS Period 4	2234384.4	3719578.2	93597.1
RCS Period 5	2233716.4	3700853.4	93095.3
RCS Period 6	2227924.3	3621657.1	92375.5
RCS Period 7	2230300.3	3635107.2	92321.2
RCS Period 8	2227552.2	3590195.5	91960.0
RCS Period 9	2231124.4	3625548.8	92060.8
RCS Period 10	2232218.7	3629717.4	92045.2

Table 1. RCS Storage Costs

6 Conclusions

We have described the design and implementation of Wayback, a comprehensive versioning file system for Linux. Wayback is implemented as a user-level file system using FUSE. When running on top of the standard Linux ext3 file system, its overhead is quite low for common modes of use.

We are considering several extensions and applications for Wayback. First, if the underlying file system does not support transactional writes, they could be forced by Wayback through sync operations. Second, it appears that a file system that never garbage collects its undo log would naturally perform very well when running on top of, or incorporated into a log-structured file system [Log]. Third, if Wayback used an undo/redo log, it would be straightforward to go forward in time as well as backward. Fourth, hierarchical version numbers and undo/redo logging would permit branching. Of course, it is not clear whether it would be any less painful to handle merging in the file system than in, say, CVS. Finally, given undo/redo logs and version numbers, keeping large files synchronized among multiple sites would be simplified – we would have only to transfer the redo log records that the remote log did not already have and then redo them. For situations where a single large file migrates among multiple sites but is accessed at one site at a time – virtual machine image migration for example – such synchronization might prove to provide dramatically faster migration times.

Availability

Wayback is publically available (under the GPL) from http://sourceforge.net/projects/wayback.

References

[3DFS] D.G. Korn and E. Krell. The 3-D File System. In *Proc. of the USENIX Summer Conference,* pp. 147-156, 1989.

[AFS] J. J. Kistler and M. Satyanarayanan. Disconnected operations in the Coda file system. In *Proc. of the 13th ACM Symposium on Operating Systems Principles.* October, 1991.

[Andrew] J. Howard, et al, *Scale and Performance in a Distributed File System,* Transactions on Computer Systems, Volume 6, February 1988.

[Bonnie] T. Bray, *The Bonnie Benchmark*, http://www.textuality.com/bonnie

[CEDAR] D. K. Gifford, R.M. Needham, and M.D. Schroeder. The Cedar File System. *Communication of the ACM*, 31(3):288-298, 1988.

[CVS] B. Berliner and J. Polk. Concurrent Versions Systems (CVS). http://www.cvshome.org .2001.

[CVFS] C. A. Soules, G.R. Goodson, J. D. Strunk and G. R. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies,* 2003

[Elephant] D. S. Santry, M.J. Feeley, N.C. Hutchinson, A.C. Veitch, R.W. Carton and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proc. of the 17th ACM Symposium on Operating System Principles.* December, 1999.

[Ext3Cow] Z. N. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadata for a Time-Shifting File System. Tech. Report HSSL-2003-03, Computer Science Department, The John Hopkins University, 2003.

[FiST] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*. June 2000.

[FUSE] M. Szeredi. Filesystem in USEr space,. http://sourceforge.net/projects/avf, 2003.

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

[Petal] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. of the 7th Conference on Architectural Support for Programming Languages and Operating Systems.* 1996.

[RCS] F. Tichy, *Software Development Control Based On System Structure Description*, PhD. Thesis, Carnegie Mellon University, 1980.

[UNDO] H. Garcia-Molina, et al, *Database Systems: The Complete Book*, Chapter 17, Prentice Hall, 2002.

[VersionFS1] Muniswamy-Reddy, et al, A Versatile and User-Oriented Versioning File System, In Proc. Of the 3rd USENIX Conference on File Storage and Technologies, March, 2004.

[VersionFS2] *A Stackable Versioning File System*, http://www.fsl.cs.sunysb.edu/project-versionfs.html

[VMS] K. McCoy, *VMS File System Internals*, Digital Press, 1990.