VERSIONFS

A Versatile and User-Oriented Versioning File System

A THESIS PRESENTED

BY

KIRAN-KUMAR MUNISWAMY-REDDY

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

STONY BROOK UNIVERSITY

Technical Report FSL-03-03

December 2003

Abstract of the Thesis

Versionfs

A Versatile and User-Oriented Versioning File System

by

Kiran-Kumar Muniswamy-Reddy

Master of Science

in

Computer Science

Stony Brook University

2003

File versioning is a useful technique for recording a history of changes. Applications of versioning include backups and disaster recovery, as well as monitoring intruders' activities. Alas, modern systems do not include an automatic and easy-to-use file versioning system. Existing backup systems are slow and inflexible for users. Even worse, they often lack backups for the most recent day's activities. Online disk snapshotting systems offer more fine-grained versioning, but still do not record the most recent changes to files. Moreover, existing systems also do not give individual users the flexibility to control versioning policies.

We designed a lightweight user-oriented versioning file system called *Versionfs*. Versionfs works with any file system, whether local or remote, and provides a host of user-configurable policies: versioning by users, groups, processes, or file names and extensions; version retention policies by maximum number of versions kept, age, or total space consumed by versions of a file; version storage policies using full copies, compressed copies, or deltas. Versionfs creates file versions automatically, transparently, and in a file-system portable manner—while maintaining Unix semantics. A set of user-level utilities allow administrators to configure and enforce default policies; users are able to set policies within configured boundaries, as well as view, control, and recover files and their versions. We have implemented the system on Linux. Our performance evaluation demonstrates overheads that are not noticeable by users under normal workloads.

To Pa and Ma

Contents

List of Figures vii

List of Tables vii

Acknowledgments ix

1 Introduction 1

2 Background 3

3 Design 6

3.1 Version Creation . . . 7

3.2 Storage Policies . . . 9

3.2.1 Full Mode . . . 9

3.2.2 Compress Mode . . . 10

3.2.3 Sparse Mode . . . 10

3.3 Retention Policies . . . 13

3.4 Manipulating Old Versions . . . 14

3.4.1 Version-Set Stat . . . 15

3.4.2 Recover a Version . . . 15

3.4.3 Open a Raw Version File . . . 15

3.4.4 Manipulation Operations . . . 16

3.4.5 Operational Scenario . . . 16

3.5 Version Cleaner and Converter . . . 17

4 Implementation 19

4.1 Copy-on-change . . . 19

4.2 Version on `mmap`'ed write . . . 19
4.3 `unlink` Operation . . . 20
4.4 `rename` Operation . . . 20
4.5 Version file visibility . . . 21
4.6 Sparse Meta-data . . . 22

5 Evaluation **24**
5.1 Feature Comparison . . . 24
5.2 Performance Comparison . . . 27
5.2.1 Configurations . . . 28
5.2.2 Workloads . . . 28
5.2.3 Am-Utils Results . . . 29
5.2.4 Postmark Results . . . 30
5.2.5 Am-Utils Copy Results . . . 32
5.2.5.1 Number . . . 33
5.2.5.2 Space . . . 34
5.2.6 Recover Benchmark Results . . . 35

6 Conclusions **37**
6.1 Future Work . . . 37

List of Figures

3.1 Recursive removal of a directory 8

3.2 Full versioning 10

3.3 Compress versioning 10

3.4 Sparse versioning - Initial state 11

3.5 Sparse version creation A 12

3.6 Sparse version creation B 12

3.7 Sparse version creation C 13

4.1 Sequence of calls for filldir 21

4.2 Sparse Meta-data 22

5.1 Am-Utils Compilation results 30

5.2 Postmark results 31

5.3 Am-Utils Copy Results - Number = 64 32

5.4 Am-Utils Copy Results - space = 500KB 34

List of Tables

5.1 Feature comparison 25
5.2 Am-Utils results 29
5.3 Postmark results 31
5.4 Am-Utils Copy Results - Number = 64 33
5.5 Am-Utils Copy Results - space = 500KB 35
5.6 Recover results 35

Acknowledgments

To my adviser Dr. Erez Zadok, for the two great years I have had here, for always standing by me and for always wanting the best for me. To Dr. Tzi-cker Chiueh and Dr. R. Sekar, for being on my committee and being generous with their inputs and comments on the work. A section like this would be incomplete without a word about my lab mates at FSL CS 2214. To Andrew P. Himmer for coming up with this idea and his contribution to the work. Charles “Chip” Wright for his valuable inputs and help throughout. Akshat Aranya for his help with the FAST submission. And finally, to all the inmates of FSL, including Dr. Erez Zadok, for making it such a great place to work and learn.

This work was partially made possible thanks to an NSF CAREER award EIA-0133589, an NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

Chapter 1

Introduction

Versioning is a technique for recording a history of changes to files. This history is useful for restoring previous versions of files, collecting a log of important changes over time, or to trace the file-system activities of an intruder. Ever since Unix became popular, users have desired a versatile and simple versioning file system. Simple mistakes such as accidental removal of files (the infamous "`rm *`" problem) could be ameliorated on Unix if users could simply execute a single command to undo such accidental file deletion. CVS is one of the most popular versioning tools [1]. CVS allows a group of users to record changes to files in a source repository, navigate branches of those changes, and recover any version officially recorded in a CVS repository. However, CVS is an interactive tool designed for shared development; it does not work transparently with all applications. Another form of versioning are backup tools such as Legato's Networker [10] or Veritas's Backup Exec [21] and FlashSnap [22]. Modern backup systems include specialized tools for users to browse a history of file changes and to initiate a recovery of backup file versions. However, backup systems are cumbersome to use, run slowly, and they do not integrate transparently with all user applications. Worse, backup periods are usually set to once a day, so potentially all file changes within the last 24-hour period are not backed up.

The Tops-20 OS included native file system versioning [3]. All user utilities worked with file versions and were able to list, browse, recover, or set versioning policies. NetApp's SnapMirror product provides automated whole file system snapshotting and the ability to recover files [12]. The Elephant file system [18] integrates block-level versioning into an FFS-like file system on FreeBSD 2.x, and includes smart algorithms for determining which versions to keep. Finally, CVFS is a comprehensive versioning file system that can version every data or meta-data change to the file system [19, 20]. CVFS's express goal is to provide secure audit-quality versioning for computer forensics applications. These file system versioning solutions suffer from one or more problems: they are not portable or available on commodity systems, they provide coarse-grained versioning, and they do not empower users to control versioning policies.

Developing a native versioning file system from scratch is a daunting task and will only work for one file system. Instead, we developed a layered, stackable file system called *Versionfs*. Versionfs can easily operate on top of any other file system and transparently add versioning functionality without modifying existing file system implementations or native on-media structures. Versionfs monitors relevant file system operations resulting from user activity, and creates backup files when users modify files. Version files are automatically hidden from users and are handled in a Unix-semantics compliant manner.

To extend the most flexibility to users and administrators, Versionfs supports various *retention* and *storage* policies. Retention policies determine how many versions to keep per file: a maximum number of most recent versions, a maximum age for the oldest version, or by the total space consumed by all versions. Storage policies determine how versions are stored: keeping full copies of old file versions, compressing old versions, or storing only block differences between versions. We define the term *version set* to mean a given file and all of its versions. A user-level dynamic library wrapper allows users to operate on a file or its version set without modifying existing applications such as `ls`, `rm`, or `mv`. Our library makes version recovery as simple as opening an old version with a text editor. All this functionality removes the need to modify user applications and gives users a lot of flexibility to work with versions.

We developed a prototype of the Versionfs system under Linux. We evaluated Versionfs under various workloads. Our user-like workloads show small performance overheads that are not noticeable by users. Our I/O intensive workloads show a performance which is comparable or better than other systems under the same workloads [19, 20].

The rest of this thesis is organized as follows. Chapter 2 surveys background work. Chapter 3 describes the design of our system. We discuss interesting implementation aspects in Chapter 4. Chapter 5 evaluates Versionfs's features, performance, and space utilization. We conclude in Chapter 6 and suggest future directions.

Chapter 2

Background

Versioning was provided by some early file systems like the Cedar File System (CFS) [6] and 3D File System (3DFS) [9]. The main disadvantage of these systems was that they were not completely transparent. Users had to create versions manually by using special tools or commands.

In CFS the users had to copy files to local disk and edit it there. To version it they had to transfer the file to a remote server and this would create a new immutable version on server. CFS is also not portable as it works only with a particular file system called *File*.

3DFS provides a library that can be used for versioning and Version Control System (VCS) is a prototype that is built to demonstrate the usefulness of the library. Versioning in VCS has to be done with explicit commands like `checkin`, `checkout`, etc. A version file is conceptually organized like a directory, but it returns a reference to only one of the files in the directory. To decide which version must be selected, each process keeps a version-mapping table that specifies the rules of version selection of a file. But to use this, all programs have to be linked with libraries provided. Another disadvantage of this system is that directories are not versioned.

CVS [1] is a popular user-land tool that is used for source code management. It is also not transparent as users have to execute commands to create and access versions. Rational ClearCase [16] is another user-land tool that is used for source code management. But it requires an administrator dedicated to it and is also expensive.

Snapshotting or check-pointing is another approach for versioning. Incremental or whole snapshots of the file system are made periodically. The checkpoints are available on-line and users can access old versions of files. Snapshotting systems include Write Anywhere File Layout(WAFL) [7], Episode [2], Venti [15], a redesigned 3DFS system [17] and Ext3cow [14].

WAFL creates read-only snapshots of the entire file system. It uses copy-on-write to avoid duplicating disk blocks that are common between a snapshot and the active file system. Users can access old versions through a `.snapshot` directory in every directory. Episode is similar to WAFL only less efficient. It copies the entire inode tree for each snapshot whereas in WAFL the data structures are designed such that to make a snapshot, only a copy of the root inode has to be made.

Venti is a storage system in which the blocks are identified by a unique hash of the block's contents. This eliminates duplicate blocks in the file system. This system can be exploited for building an efficient snapshotting system.

In the redesigned 3DFS, A network file server that uses an optical disk jukebox is used to store the snapshots. A user level program finds all the files that have been modified since the last time and sends them to the server. The server writes them to an optical disk. It provides a read-only access to the previous versions of the file, with a `file@date` syntax. Another disadvantage is that the optical disk jukebox takes about 15 seconds to switch disks and this affects the access time.

Ext3cow extends ext3 to support versioning by changing some of ext3's meta-data structures. It offers a `file@epoch` interface for accessing older versions. However the epoch is one unsigned long number which is not very convenient for users. Ext3cow also extends the inode structure to store previous version inode number. This way traversing through each version can be implemented efficiently.

Snapshotting systems have several disadvantages, the largest one being that users cannot undo changes made to a file between snapshots. Snapshotting systems treat all files equally. This is a disadvantage because all files do not have the same usage pattern and may have different storage and retention requirements. When space must be recovered, whole snapshots must be purged. Often, managing such snapshot systems requires the intervention of the administrator.

Versioning with copy-on-write is another technique that is used by Tops-20 [3], VMS [11], Elephant File System [18], and CVFS [19, 20]. Though Tops-20 and VMS had automatic versioning of files, they did not handle all operations, such as rename, etc.

Elephant is implemented in the FreeBSD 2.2.8 kernel. Elephant transparently creates a new version of a file on the first write to an open file. Elephant also provides users with four retention policies: keep one, keep all, keep safe, and keep landmark. Keep one is no versioning, keep all retains every version of a file, keep safe keeps versions for a specific period of time but does not retain the long term history of the file, and keep *landmark* retains only important versions in a file's history. A user can mark a version as a landmark or the system uses heuristics to mark other versions as landmark versions. Elephant also provides users with the ability to register their own space reclamation policies. However, Elephant has its own low-level disk format and cannot be used with other systems. It also lacks the ability to provide an extension list to be included or excluded from versioning. User level applications have to be modified to access old versions of a file.

CVFS is a system designed with security in mind. Each individual write or the smallest meta-data change (including atime updates) results in a new version. Since many versions are created, new data structures were designed so that old versions can be stored and accessed efficiently. As CVFS was designed for security purposes, it does not have facilities for the user to access or customize versioning.

Clearly, attempts were made to make versioning a part of the OS. However, modern operating systems still do not include versioning support. We believe that these past attempts were not very successful as they were not convenient or flexible enough for users. A feature comparison of the different versioning systems is made in Section 5.1.

Chapter 3

Design

When designing Versionfs, we emphasized ease-of-use and flexibility. We had the following four goals:

- **Easy-to-use** We designed our system such that a single user could easily use it as a personal backup system. This meant that we chose to use per-file granularity for versions, because users are less concerned with entire file system versioning or block-level versioning. Another requirement was that the interface would be simple. For normal operations, Versionfs should be completely transparent.

- **Flexibility** While we wanted our system to be easy to use, we also made it flexible by providing the user with options. The user can select minimums and maximums for how many versions to store and additionally how to store the versions. The system administrator can also enforce default, minimum, and maximum policies.

- **Portability** Versionfs provides portable versioning. The most common operation is to read or write from the *current version*, or the file that would exist even if Versionfs was not being used. We implemented Versionfs as a kernel-level file system so that applications do not need any modifications for accessing the current version. All applications can access the current version without any changes, but for efficiency we do not overload the system call interfaces for accessing previous versions. For previous version access, we instead use library wrappers, so that the majority of applications do not require any changes. Additionally no operating system changes are required for Versionfs.

- **Efficiency** There are two ways in which we approach efficiency. First, we need to maximize current version performance. Second, we want to use as little space as possible for versions to allow a deeper history to be kept. These goals are often conflicting, so we provide various storage and retention policies to users and system administrators.

We chose a stackable file system to balance efficiency and portability. Stackable file systems run in kernel-space and perform well. For current version access, this results in a low overhead. Stackable file systems are also portable. System call interfaces remain unchanged so no application modifications are required.

The rest of this section is organized as follows. Section 3.1 describes how versions are created. Section 3.2 describes *storage policies*, which determine the internal format of version files. Storage policies include full copies, compressed copies, and block deltas using sparse files. Section 3.3 describes *retention policies*, which determine how our system decides which versions to retain and which versions to remove. Section 3.4 describes how previous versions are accessed and manipulated through a set of `ioctl`s and a library. Section 3.5 describes our version cleaning daemon.

3.1 Version Creation

In Versionfs, the *head*, or current, version is stored as a regular file, so it maintains the access characteristics of the underlying file system. This design avoids a performance penalty for reading the current version. The set of a file and all its versions is called a *version set*. Each version is stored as a separate file. For example, the file `foo`'s nth version is named "`foo;`Xn". X is substituted depending on the storage policy used for the version. X could be: "`f`" indicating a full copy, "`c`" indicating a compressed version, "`s`" indicating a sparse version, and "`d`" indicating a versioned directory. Along with each version set we store a meta-data file (e.g., "`foo;i`" that contains the minimum and maximum version numbers as well as the storage method for each version. The meta-data file acts as a cache of the directory to improve performance. This file allows Versionfs to quickly identify versions and know what name to assign to a new version. The meta-data file can be regenerated entirely from the entries provided by `readdir`. The meta-data file can be recovered because we can get the storage method and the version number from the version file names. On version creation, Versionfs also discards older versions according to the retention policies defined in Section 3.3.

Newly created versions are created using a *copy-on-change* policy. *Copy-on-change* differs from copy-on-write in that writes that do not modify data will not cause versions to be created. There are six types of operations that create a version: writes (either through `write`(2) or `mmap` writes), `unlink`, `rmdir`, `rename`, `truncate`, and ownership or permission modifications.

The write operations are intercepted by our stackable file system. If there are differences between the existing data and the new data, then Versionfs creates a new version. Between each open and close, only one version is created. This heuristic approximates one version per save, which is intuitive for users. The Elephant file system uses a similar one-version-per-save heuristic [18].

`unlink` also creates a version. For some version storage policies (e.g., compression), `unlink` results in the file's data being copied. If the storage policy permits, then `unlink` is translated into a `rename` operation to improve performance. When `unlink` can be translated into a `rename`, we reduce the amount of I/O required for version creation.

`rmdir` is converted into a rename of `foo` to "`foo;d1`". We only rename a directory that appears to be empty from the perspective of a user. To do this we execute a `readdir` operation to ensure that all files are versions or the version set meta-data file.

Figure 3.1 shows a tree before and after it is removed by `rm -rf`. `rm` operates in a depth first manner. First `rm` descends into `A` and calls `unlink(b)`. To create a version for `b`, Versionfs instead renames `b` to `b;f1`. Next, `rm` descends into `C`, and `d` and `e` are versioned the same way `b` was. Next, `rm` calls `rmdir` on `C`. Versionfs uses `readdir` to check that `C` does not contain any files visible to the user, and then renames it to `C;d1`. Finally, `A` is versioned by renaming it to `A;d1`.

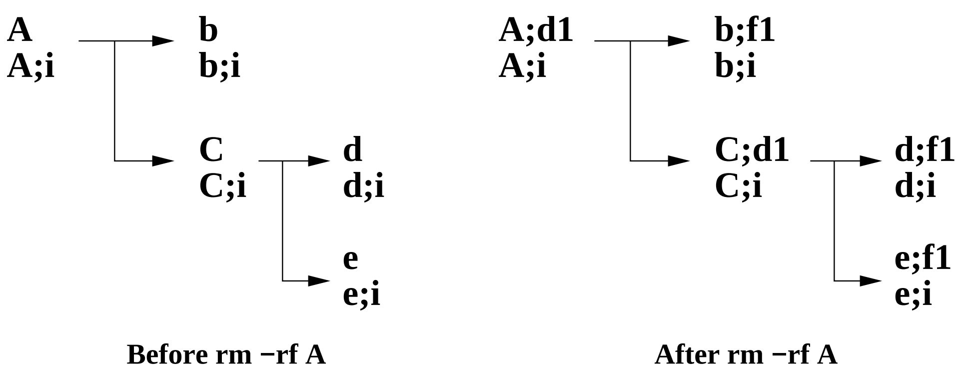

Figure 3.1: rm -rf on directory A

`rename` must create a version of the source file and the destination file. The source file needs a version so that the user can recover it later using the source name. If the destination file exists, then it too must be versioned so its contents are preserved. We treat the new file and the old file as separate version sets.

`truncate` must also create a version. File truncation is divided into two cases: truncating to a non-zero size and truncating to zero bytes. When truncating a file to a non-zero size, Versionfs must make a version and copy data for all pages that would be discarded by the truncation operation. However, when truncating a file to zero bytes, Versionfs does not need to copy the data. Versionfs translates the `truncate` into a `rename` similar to our handling of `unlink`. Versionfs then recreates the empty file. This saves on I/O that would be required for the copy.

File meta-data is modified when owner or permissions are changed, therefore `chmod` and `chown` also create versions. This is particularly useful for security applications. If the storage policy permits (e.g., sparse mode), then no data is copied.

3.2 Storage Policies

Storage policies define our internal format for versions. The system administrator sets the default policy, which may be overridden by the user. We have developed three policies: full, compressed, and sparse mode.

3.2.1 Full Mode

Full mode makes an entire copy of the file each time a version is created. As can be seen in Figure 3.2, each version is stored as a separate file of the form "`foo;f`*N*", where N is the version number. The current, or *head*, version is `foo`. The oldest version in the diagram is "`foo;f8`". At the time version 8 was created, its contents were located in `foo`. When the page A2 was written over the page A1, Versionfs copied the entire head version to a backup, "`foo;f8`". After the backup copy was made, A2 was written to `foo`, then B1, C2, and D2 were written without any further version creation. This demonstrates that in full mode, once the version is initially created, there is no additional overhead for read or write. The creation of version 9 proceeds similarly to the creation of version 8. The first write attempted to overwrite the contents of page A2, with a page that had the same contents. Versionfs compared the new page to the old page. Versionfs noticed the two pages were the same, and hence bypassed version creation. When page B2 overwrote page B1, the contents of `foo` were copied to "`foo;f9`", and further writes directly modified `foo`. Pages C2, D3, and E1 where directly written to the head version. Version 10 was created in the same way. Writing A2 and B2 would not have caused a version to be created, but writing C3 over C2 would have copied the head version to "`foo;f10`". Finally, a truncate command was issued. In the same session, C3 overwrote C2 to create a version, so there was no need to make a new version and the truncation directly modified the head version.

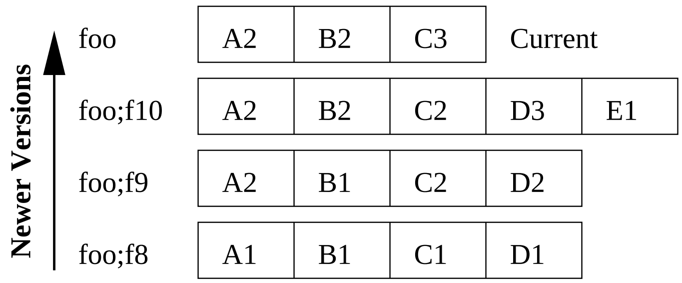

Figure 3.2: Full versioning. Each version is stored as a complete copy and each rectangle represents one page.

3.2.2 Compress Mode

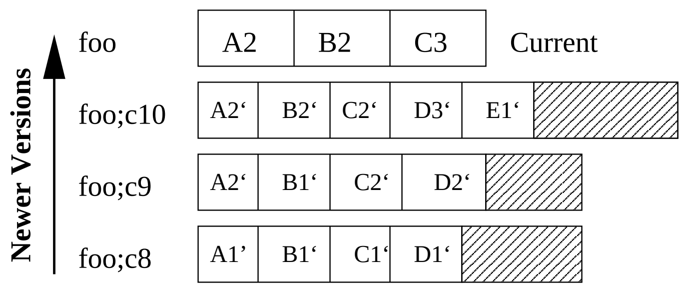

Figure 3.3: Compress versioning. Each version is stored as a compressed copy. Each rectangle represents one compressed page. Rectangles with hatch pattern indicate space saved due to compression

Compress mode is the same as full mode, except that the copies of the file are compressed as shown in Figure 3.3. Space can only be saved in units of disk blocks. If the original file size is less than one block, then Versionfs does not use compression because there is no way to save any space. Compress mode reduces space utilization and I/O wait time, but requires more system time. Versions can also be converted to compress mode offline using our cleaner described in Section 3.5.

3.2.3 Sparse Mode

When holes are created in files (e.g., through `lseek` and `write`), file systems like Ext2, FFS, and UFS do not allocate blocks. Files with holes are called *sparse* files. Sparse mode versioning stores only block deltas between two versions. Only the blocks that change between versions are saved in the version file. It uses sparse files on the underlying file system to save space. Compared to full mode, sparse mode versions reduce the amount of space used by versions and the I/O time. The semantics of sparse files are that when a sparse section is read, a zero-filled page is returned. There is no way to differentiate this type of page with a page that is genuinely filled with zeros. To identify which pages are holes in the sparse version file, Versionfs stores sparse version meta-data information at the end of the version file. The meta-data contains the original size of the file, and a bitmap that records which pages are valid in this file. Versionfs does not preallocate the intermediate data pages, but does leave logical holes. These holes allow Versionfs to backup changed pages on future writes, without costly data shifting operations [23].

Two important properties of our sparse format are: (1) a normal file can be converted into a sparse version by renaming it and then appending a sparse header, and (2) we can always discard tail versions because reconstruction only uses more recent versions.

To reconstruct version N of a sparse file `foo`, Versionfs first opens “`foo;sN`”. Versionfs reconstructs the file one page at a time. If a page is missing from “`foo;sN`”, then we open the next version and attempt to retrieve the page from that version. We repeat this process until the page is found. This procedure always terminates, because the head version is always complete.

We will present an identical situation to that of Figure 3.2, but instead of using full mode, we will use sparse mode. We first describe each operation step by step in Figures 3.4, 3.5, and 3.6. Figure 3.7 shows the end result of the operations.

Figure 3.4 shows the contents of `foo` when no versions exist. A meta-data file, “`foo;i`”, which contains the next version number also exists. The first version created in this sequence is version 8.

foo	A1	B1	C1	D1	Current

Figure 3.4: Sparse versioning: Only `foo` exists.

As seen in Figure 3.5, to create version 8, the page A2 is written over the contents of A1. The head version is named `foo` and a new file “`foo;s8`” is created. “`foo;s8`” only contains sparse meta-data (the original size and bitmap). A1 is written to “`foo;s8`”, the sparse bitmap is updated, and finally A2 is written to `foo`. Unlike full mode, no other data blocks are copied to “`foo;s8`” yet and “`foo;s8`” remains open. The next write attempts to overwrite the contents of page B1 with the same data. Because there are no differences in these two pages, Versionfs does not write the block to the sparse file. Next, C2 overwrites C1 and Versionfs writes C1 to the sparse file before writing C2 to the head version. Versionfs additionally updates the sparse meta-data bitmap. Page D is written in the same way as page C.

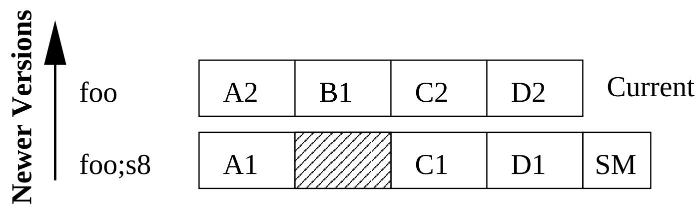

Figure 3.5: Sparse versioning: Creation of version 8. Each version stores only block deltas, and each rectangle represents one page. Rectangles with hatch patterns are sparse. Sparse meta-data is represented by the rectangle with "SM" inside. The SM block is less than one page and may be part of the last short page of the file.

Figure 3.6 shows the creation of version 9. Version 9 is created in a similar manner to version 8. First, a write attempts to overwrite page A2 with the same contents which does not result in a version. Next, the page B2 attempts to overwrite B1. Versionfs now creates "`foo;s9`" with no data except the sparse meta-data. B1 is copied into "`foo;s9`" and B2 is written to `foo`. The application then attempts to overwrite page C2 with the same data. The page is not copied into the version file because Versionfs compares them before writing the data. The page D3 overwrites the page D2, and D2 is copied into "`foo;s9`". Finally, E1 is written to the head version. Because E1 is extending the file, it is not overwriting any data, so there is no need to copy it into the version.

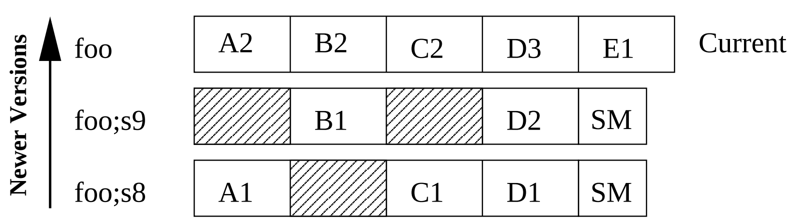

Figure 3.6: Sparse versioning: Creation of version 9. Each version stores only block deltas, and each rectangle represents one page. Rectangles with hatch patterns are sparse. Sparse meta-data is represented by the rectangle with "SM" inside.

The last version in this sequence is version 10. The creation of version 10 proceeds similarly to the creation of versions 8 and 9. The pages A2, B2, and C3 are written to the head version. Only C3 differs from the previous contents of the file, so only C2 is written to the version file, "`foo;s10`".

Next, the file is truncated to 12KB, so D3 and E1 need to be copied into "`foo;s10`". The result of these operations can be seen in Figure 3.7.

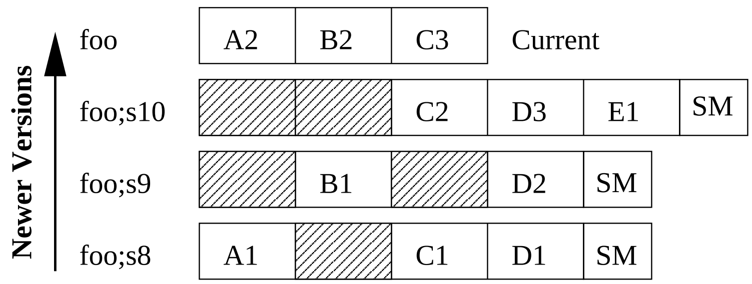

Figure 3.7: Sparse versioning: Creation of version 10. Each version stores only block deltas, and each rectangle represents one page. Rectangles with hatch patterns are sparse. Sparse meta-data is represented by the rectangle with "SM" inside.

3.3 Retention Policies

We have developed three version retention policies:

- **Number** The user can set the maximum and minimum number of versions in a version set. This policy is attractive because at least some history is always kept.
- **Time** The user can set the maximum and minimum amount of time to retain versions. This allows the user to ensure that a history exists for at least a certain period of time.
- **Space** The user can set the maximum and minimum amount of space that a version set can consume. This policy allows a deep history tree for small files, but does not allow one large file to use up too much space.

A version is never discarded if discarding it would violate a policy's minimum. The minimum values take precedence over the maximum values. If a version set does not violate any policy's minimum and the version set exceeds any one policy's maximum, then versions are discarded beginning from the tail of the version set.

Providing a minimum and maximum version becomes useful when a combination of two policies are used. For example, a user can specify that the number of versions to be kept should be 10–100 and 2–5 days of versions should be kept. This policy ensures that both the 10 most recent versions and at least two days of history is kept. Minimum values ensure that versions are not prematurely deleted, and maximums specify when versions should be removed.

Each user and the administrator can set a separate policy for each file size, file name, file extension, process name, and time of day. File size policies are useful because they allow the user to ensure that large files do not use too much disk space. File name policies are a convenient method of explicitly excluding or including particular files from versioning. File extension policies are useful because file names are highly correlated with the actual file type [4]. This type of policy could be used to exclude large multimedia files or regenerable file such as `.o` objects. Process name can be used to exclude or include particular programs. A user may want any file created by a text editor to be versioned, but to exclude files generated by their Web browser. Time-of-day policies are useful for administrators because they can be used to keep track of changes that happen outside of business hours or other possibly suspicious times.

For all policies, the system administrator can provide defaults, minimum, and maximum values that can be customized by the user (to enforce a specific policy, the system administrator can set the default, minimum, and maximum values to the same value). In case of conflicts, administrator-defined values override user-defined values.

3.4 Manipulating Old Versions

By default, users are allowed to read and manipulate their own versions, though the system administrator can turn off read or read-write access to previous versions. Turning off read access is useful because system administrators can have a log of user activity without having the user know what is in the log. Turning of read-write access is useful because users cannot modify old versions either intentionally or accidentally.

Versionfs exposes a set of `ioctl`s to user space programs, and relies on a library that we wrote, *libversionfs* (LVFS) to convert standard system call wrappers into Versionfs `ioctl`s. LVFS can be used as an `LD_PRELOAD` library that intercepts each library system call wrapper and directory functions (e.g., `open`, `rename`, or `readdir`). After intercepting the library call, LVFS determines if the user is accessing an old version or the current version (or a file on a file system other than Versionfs). If a previous version is being accessed, then LVFS implements the desired function in terms of Versionfs `ioctl`s; otherwise the standard library wrapper is used. The use of an `LD_PRELOAD` wrapper greatly simplifies the kernel code; as versions are not directly accessible through standard VFS methods.

Versionfs provides the following `ioctl`s: version set stat, recover a version, open a raw version file, and also several manipulation operations (e.g., `rename` and `chown`). Each `ioctl` takes the file descriptor of a parent directory within Versionfs. When a file name is used, it is a relative path starting from that file descriptor.

3.4.1 Version-Set Stat

Version-set stat (`vs_stat`) returns the minimum and maximum versions in a version set and the storage policy for each version. `vs_stat` also returns the same information as `stat` for each version.

3.4.2 Recover a Version

The version recovery `ioctl` takes a file name F, a version number N, and a destination file descriptor D as arguments. It writes the contents of F's N-th version to the file descriptor D. Providing a file descriptor is useful because it gives the application programmer a great deal of flexibility. If D is a file, then Versionfs copies the version to that file. If the file position was located at the end, then Versionfs appends the version to the file. If D is a socket, then Versionfs streams the version over the network. A previous version of the file can even be recovered to the head version. In this case, version creation takes place as normal.

This `ioctl` is used by LVFS to `open` a version file. To preserve the version history, Version files can be opened for reading only. LVFS recovers the version to a temporary file, re-opens the temporary file read-only, unlinks the temporary file, and returns the read-only file descriptor to the caller. After this operation, the caller has a file descriptor which can be used to read the contents of a version.

3.4.3 Open a Raw Version File

Opening a raw version returns a file descriptor to a raw version file. Users can only read raw versions, but root can read and write versions. This `ioctl` is used to implement `readdir` and for our version cleaner and converter. The application must first run version-set stat to determine what the version number and storage policy of the file are. Without knowing the corresponding storage policy, the application can not interpret the format of the version file correctly.

Through the normal VFS methods, version files are hidden from user space, therefore when an application calls `readdir` it will not see deleted versions. When the application calls `readdir`, LVFS runs `readdir` on the current version of the raw directory so that deleted versions are returned to user space. LVFS then interprets the contents of the underlying directory to present a consistent view to user space. Additionally, deleted directories cannot be opened through standard VFS calls, therefore we use the raw open `ioctl` to access them as well.

3.4.4 Manipulation Operations

We also provide `ioctl`s that `rename`, `unlink`, `rmdir`, `chown`, and `chmod` an entire version set. For example, the version-set chown operation modifies the owner of each version in the version set. To ensure atomicity, Versionfs locks the directory while performing version-set operations. The standard library wrappers simply invoke these version set manipulation `ioctl`s. The system administrator can disable these `ioctl`s so that previous versions are not modified.

3.4.5 Operational Scenario

LVFS exposes all versions of files. For example, version 8 of `foo` is presented as "`foo;8`" regardless of the underlying storage policy. A user can read old versions simply by opening them. When a manipulation operation is performed on `foo`, then all files in `foo`'s version set are manipulated.

An example session using LVFS is as follows. Normally the user sees only the head version, in this case `foo`.

```
$ echo -n Hello > foo
$ echo -n ", world" >> foo
$ echo '!' >> foo
$ ls
foo
$ cat foo
Hello world!
```

Next, the user sets an LD_PRELOAD in order to see all versions.

```
$ LD_PRELOAD=libversionfs.so
$ export LD_PRELOAD
```

After using LVFS as an LD_PRELOAD, the user sees all versions of `foo` in directory listings and can then access them. Regardless of the underlying storage format, LVFS presents a consistent interface. The third version of `foo` is named "`foo;2`". There are no modifications required to standard applications.

```
$ ls
foo  foo;1  foo;2
```

If the user wants to examine a version, all they need to do is open it. Any dynamically linked program that uses the library wrappers to system calls can be used to view older versions. For example, `diff` can be used to examine the differences between a file and an older version.

```
$ cat 'foo;1'
Hello
$ cat 'foo;2'
Hello, world
$ diff foo 'foo;1'
1c1
< Hello, world!
---
> Hello
```

LVFS can also be used to modify an entire version set. For example, if a user wants to rename every version in the version set, all they need to do is use the standard `mv` command.

```
$ mv foo bar
$ ls
bar  bar;1  bar;2
```

3.5 Version Cleaner and Converter

Using the version-set stat and open raw `ioctl`s we have implemented a version cleaner and converter.

As new versions are created, Versionfs prunes versions according to the retention policy as defined in Section 3.3. Versionfs cannot implement time-based policies entirely in the file system. For example, a user may edit a file in bursts. At the time the versions are created, none of them exceed the maximum time limit. However, after some time has elapsed, those versions can be older than the maximum time limit. Versionfs does not evaluate the retention policies until a new version is created. To account for this, the cleaner uses the same retention policies to determine which versions should be pruned. Additionally, the cleaner can convert versions to more compact formats (e.g., compressed versions) rather than removing older versions.

The cleaner is also responsible for pruning directory trees. We do not prune directories in the kernel because recursive operations are too expensive to run in the kernel. Additionally, if directory trees were pruned in the kernel, then users would be surprised when seemingly simple operations took a significantly longer time than expected. This could happen, for example, if a user writes to a file that used to be a directory. If the user's new version needed to discard the entire directory, then the user's simple operation would take an inexplicably large amount of time.

Finally, the version cleaner can also be used as a high-level file system checker (i.e., `fsck`) to reconstruct damaged or corrupted version meta-data files and recover sparse meta-data blocks.

Chapter 4

Implementation

We implemented a prototype of Versionfs on Linux 2.4.20 starting from a stackable file system template. Versionfs is 12,476 lines of code. Out of this, 2,844 lines were for the various `ioctl`s that we implemented to recover, access, and modify versions. Excluding the code for the `ioctl`s, this is an addition of 74.9% more code to the stackable file-system template that we used. We implemented all the features described in our design, but we do not yet version hard links.

4.1 Copy-on-change

The stackable file system templates we used cache pages both on the upper-level and the lower-level file system. We take advantage of this double buffering so that the Versionfs file can be modified by the user (through `write` or `mmap`), but the underlying file is not yet changed. In `commit_write` (used for `write`(2) calls) and `writepage` (used for `mmap`'ed writes), Versionfs compares the contents of the lower-level page to the contents of the upper-level page. If they differ, then the contents of the lower-level page are used to save the version. The memory comparison does not increase system time significantly, but the amount of data versioned, and hence I/O time are reduced significantly.

4.2 Version on `mmap`'ed write

A file must be versioned on the first `mmap`'ed write that modifies its contents. In order to create the version we need to know the name of the file. However, on `mmap`'ed writes, the kernel provides us with only a pointer to the page containing the modified data. Although we can derive the inode from the pointer to the page, we cannot obtain the name or the path from the page.

As a workaround, we grab a reference to the lower-level dentry and store it in the upper-level inode at the point where the lower-level and the upper-level inodes are interposed. The dentry gives us the name required for version creation during `mmap` write. The reference to the lower-level dentry is released when the upper-level inode itself is released.

4.3 `unlink` Operation

An `unlink` creates a version of the file. In FULL and SPARSE modes, instead of creating a new version, we `rename` the head (original) file appropriately. In SPARSE mode, apart from renaming the files, we also append the appropriate meta-data to the end of the file. This optimization drastically reduces the amount of I/O required, especially when large number of files must be unlinked. Such an optimization is not possible for COMPRESS mode. This is because in COMPRESS mode, the version file must be compressed, whereas the file being unlinked is not in the compressed form.

Since an `unlink` operation never frees up space, we provide an `ioctl` in LVFS that can be used to delete a version set completely.

4.4 `rename` Operation

On running `rename foo bar`, when both are regular files, the following cases are possible:

1. **`bar` does not exist:** In this case, we need to create a version of `foo`, followed by a call to `vfs_rename`. If both these phases succeed, then we have to create the meta-data file `bar;i` for `bar`.

2. **`bar` does not exist but `bar;i` exists:** This happens if `bar` has already been deleted and so its versions and meta-data files are left behind. In such a case, after the call to `vfs_rename`, we need to open `bar;i`, associate it with the newly renamed inode and use the storage policy, minimum, and maximum version numbers defined in `bar;i`.

3. **`bar` exists:** If `bar` already exists, we need to create a version of `bar` before calling `vfs_rename`. After `vfs_rename` has been called, we need to swap the `;i` file pointers and also the maximum and minimum version numbers of `foo` and `bar`

A call to `rename` does not rename the whole version set. We provide an `ioctl`, `versionfs_rename_ioctl`, as part of LVFS. Using `versionfs_rename_ioctl`, the whole version set can be renamed (i.e., even the `;i` and the version files).

4.5 Version file visibility

Versionfs does not allow direct access of version and meta-data files so that users would not modify them inadvertently. This is achieved by filtering them out at two points: `lookup` and `readdir`.

The function `lookup` is called when a user tries to access files for operations like `open`, `truncate`, etc. Versionfs's `lookup` operation, `versionfs_lookup`, returns -EINVAL when a user tries to access a version file. This prevents a user from accessing a version file.

The system call `getdents` lists entries in a directory. The user passes a buffer as parameter and `getdents` fills it with entries and returns the number of bytes left in the buffer. On reaching the end of directory, 0 is returned. The sequence of calls for `getdents` on Versionfs is shown in Figure 4.1. The system call `getdents` calls `filldir`, a callback function, with each entry in the directory. The callback `filldir` puts the data into the buffer and indicates to `getdents` if there is space for more entries in the buffer.

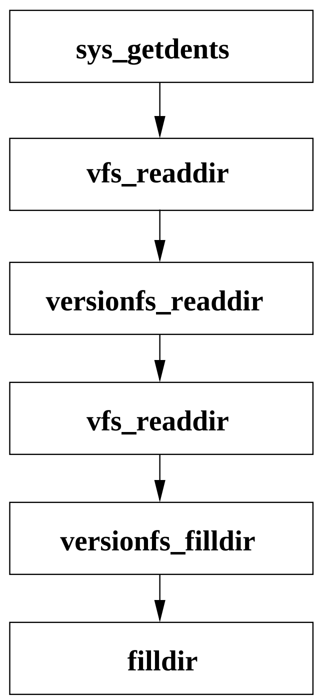

Figure 4.1: Sequence of calls for filldir

The system call `getdents` calls `vfs_readdir`, which in turn calls `versionfs_readdir`. The function `versionfs_readdir` calls the lower-level file system's `readdir` with `versionfs_filldir` as the callback function. The callback function versionfs_filldir filters out names that match either a version file or the meta-data file and calls filldir. As the function filldir is never called with version or meta-data file names, version and meta-data file names are not listed.

However, the readdir of various file systems are not consistent. Some file system's readdir calls filldir until the output buffer is full. Other file systems stop calling filldir once they have read 4KB of data from the directory, irrespective of whether the output buffer is filled or not. This causes programs, like ls or rm ,that depend on getdents to report that there are no files when there are files left in the directory.

To make Versionfs work with all file systems, in versionfs_getdents, call vfs_getdents calls repeatedly until we fill at least one entry. We perform special checking for empty directories to avoid going into an infinite loop.

4.6 Sparse Meta-data

Initially we designed our system so that whole pages are interspersed throughout a file that contain information about which pages are valid and which are sparse. For example, page 0 was one such page; it contained information as to which of the next 32,768 pages were filled or sparse. Page 32,769 was reserved to contain similar information for the next 32,768 pages. The drawback with this approach is that it adds an overhead for small files. For example, if a file is only 2KB long, it would actually occupy 6KB of storage. So instead of saving on space in the sparse mode, it would actually consume more space if all the files are small.

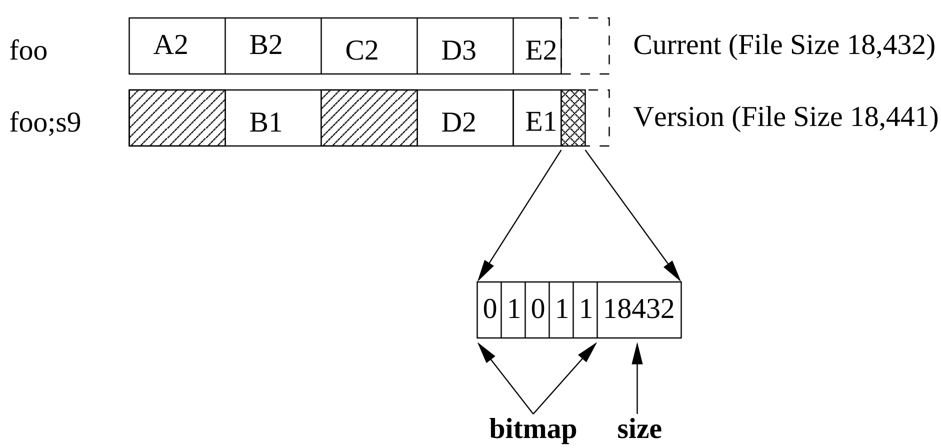

Figure 4.2: Sparse Meta-data: Rectangle with cross-hatch pattern is the meta-data. It consists of a bitmap to identify valid and invalid pages and the file size

In the approach we are currently using, the meta-data is stored at the end of each sparse file. The meta-data stored is the original size of the file and the required number of bytes to store a bitmap of valid and invalid pages in the sparse file as shown in Figure 4.2. The file size is stored at the end of the file to identify the bytes that form the bitmap. In the Figure, `foo` is the head file and `foo;s9` is the sparse version file. In `foo;s9`, pages 1, 3, and 4 are valid. The other pages 0 and 2 are sparse. The corresponding bitmap is as shown in the Figure. In this case, only 1 byte is required to store the bitmap. The size of the sparse file is 18,441 bytes. The original file was 18,432 bytes. The extra 9 bytes are to store the file size and the bitmap.

A disk block is the minimal unit of space allocation in file systems. Most files do not consume all the bytes in their last disk block. The advantage of this approach is that the meta-data can fit into the unused space in the last page for most files as shown in the Figure 4.2. Hence meta-data can be stored without any additional space overhead.

Chapter 5

Evaluation

We evaluate our implementation of Versionfs in terms of features as well as performance. In Section 5.1, we compare the features of Versionfs with some other versioning systems. In Section 5.2, we evaluate the performance of our system by executing various benchmarks.

5.1 Feature Comparison

In this section, we describe and compare the features of Ext3cow [14], Elephant [18], CVFS [19, 20], and Versionfs. We selected them because they version files without any user intervention. We do not include some of the older systems like Tops-20 [3] and VMS [11] as they did not handle operations such as rename, etc. We chose Elephant and CVFS because they create versions when users modify files rather than at predefined intervals such as WAFL [7]. We chose Ext3cow as it is a recent representative of snapshotting systems. We do not include Venti [15] as it provides a framework that can be used for versioning rather being than a versioning system in itself.

We identified the following eleven features that we have summarized in Table 5.1:

1. **Multiple storage policies:** Versionfs provides three storage policies. Users can choose the policy that suits their needs and lies within resource constraints. Users are at a disadvantage with Ext3cow, Elephant, and CVFS as they only have one storage policy.

2. **Multiple retention policies:** Apart from the way data is stored on disk, different users want data to be reclaimed in different ways. Versionfs provides three flexible retention policies. Elephant is the only other file system that provides retention policies. Elephant provides four retention policies: keep one, keep all, keep safe, and keep landmark. Keep one is no versioning, keep all retains every version of a file, keep safe retains versions long enough

	Feature	Ext3cow	Elephantfs	CVFS	Versionfs
1	Multiple Storage policies				✔
2	Multiple Retention Policies		✔		✔
3	Per-user extension inclusion/exclusion list to version				✔
4	Administrator can override user policies				✔
5	Unmodified applications can access previous versions	✔			✔
6	Works with any file system				✔
7	User-land cleaner		✔	✔	✔
8	User-land converter		✔		✔
9	Allows users to register their own reclamation policies		✔		
10	Version tagging		✔		
11	File system type	Disk based	Disk based	Disk based	Stackable

Table 5.1: Feature comparison. A check mark indicates that the feature is supported, otherwise it is not.

for recovering but does not retain the long term history of a file and keep landmark retains a long-term history of files.

3. **Per-user extension inclusion/exclusion list to version:** It is important that users have the ability to choose the files to be versioned and the policy to be used for versioning, because all files do not need to be versioned equally. Also, file names with same extensions have similar usage patterns [4]. Versionfs allows users to specify an extension list that can be used to choose the files to be versioned. For example, all `.c` and `.java` files should be versioned. At times it is more convenient to specify which files should not be versioned rather than which files should be versioned. For example, `.o` files do not need to be versioned. Keeping this in view, Versionfs allows users to specify a list of items to be excluded from versioning. Elephant allows groups of files to have the same retention policies, but does not allow explicit extension lists.

4. **Administrator can override user policies:** Providing users a lot of flexibility means users could mis-configure the policies. It is important that administrators can override or set bounds to the user policies. Versionfs allows administrators to set an upper bound and lower bound on the space, time, and number of versions retained.

5. **Unmodified application access to previous versions:** Versioning is only truly transparent to the user if previous versions can be accessed without making modifications to the regular user-level applications like `ls`, `cat`, productivity applications, etc. This is possible in Versionfs (through libversionfs) and also in Ext3cow. Elephant and CVFS need modified tools to access the previous versions of a file.

6. **Works with any file system:** Versionfs is implemented as a stackable file system, so it can be used with any file system, whereas Ext3cow, Elephant, and CVFS are disk-based file systems and cannot be used in conjunction with any other file system. The advantage of stacking is that the underlying file system can be chosen based on the needs of the user. For example, if the user is expecting a lot of files to be created, the user can stack on top of a Hash Tree based or tail-merging file system to improve performance and disk usage.

7. **User-land version cleaner:** Apart from reclaiming space while a file is being actively modified, it is important to reclaim space from files that have not been accessed in a while. A user-level cleaner has the advantage that space reclamation can be accurate, whereas in a snapshotting system whole snapshots are purged to reclaim space. Versionfs provides a

user-land daemon that reclaims space from old, unused versions of files. Ext3cow does not have a user-land cleaner whereas Elephant and CVFS have one.

8. **User-land version converter:** Converting between the different storage policies is another useful method of reclaiming space. For example, full copies of files can be compressed by the user-land converter to reclaim space, instead of deleting files. Versionfs supports this feature. Though Elephant's cleaner can compress files, unlike Versionfs the compressed files do not have a built in support in Elephant and cannot be used directly by users.

9. **Allows users to register their own reclamation policies:** Users might prefer policies other than the system default. Elephant is the only file system that currently allows users to set up custom reclamation policies.

10. **Version tagging:** Users want to mark the state of a set of files with a common name so that they can revert back to the state in the future. Elephant is the only file system that currently has this facility.

11. **File system type:** Versionfs is a stackable file system, whereas Ext3cow, Elephant, and CVFS are disk-based file systems and have their own disk layouts. Versionfs can be used over file systems such as NFS or CIFS.

5.2 Performance Comparison

We ran all the benchmarks on a 1.7GHz Pentium 4 machine with 1GB of RAM. All experiments were located on a 20GB 7200 RPM Western Digital Caviar IDE disk. The machine was running Red Hat Linux 9 with a vanilla 2.4.22 kernel. To ensure a cold cache, we unmounted the file systems on which the experiments took place between each run of a test. We ran each test at least 10 times. To reduce I/O effects due to ZCAV we located the tests on a partition toward the outside of the disk that was just large enough for the test data [5]. We recorded elapsed, system, user times, and the amount of disk space utilized for all tests. We also recorded the wait times for all tests. Wait time is mostly I/O time, but other factors like scheduling time can also affect it. We report the wait time where it is relevant or has been affected by the test. We computed 95% confidence intervals for the mean elapsed and system time using the Student-t distribution. In each case, the half-widths of the confidence intervals were less than 5% of the mean. The space used does not change between different runs of the same test. In all our benchmarks except in Am-Utils compile, the user times are very small and are not visible in the graphs. In the Am-Utils compile benchmark, the user time is not affected by Versionfs as it operates only in the kernel. Therefore we do not discuss the user times in any of our results. We also ran the same benchmarks on Ext3 as a baseline.

5.2.1 Configurations

We used the following storage policies for evaluating Versionfs:

- **FULL** Versionfs using the full policy.
- **COMPRESS** Versionfs using the compress policy.
- **SPARSE** Versionfs using the sparse policy.

We used the following retention policies for evaluating Versionfs:

- **NUMBER** Versionfs using the number policy.
- **SPACE** Versionfs using the space policy.

For all benchmarks, one storage and one retention configuration are concurrently chosen.

We did not benchmark the time retention policy as it is similar in behavior to space retention policy.

5.2.2 Workloads

We ran four different benchmarks on our system: a CPU-intensive benchmark, an I/O intensive benchmark, a benchmark that simulates the activity of a user on a source tree, and a benchmark that measures the time needed to recover files.

The first workload was a build of Am-Utils [13]. We used Am-Utils 6.1b3: it contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Though the Am-Utils compile is CPU intensive, it contains a fair mix of file system operations, which result in the creation of multiple versions of files and directories. We ran this benchmark with all storage policies that we support. We also ran this benchmark under all retention policies, but we report only one set of results as they are nearly identical. This workload demonstrates the performance impact a user sees when using Versionfs under a normal workload.

The second workload we chose was Postmark [8]. Postmark simulates the operation of electronic mail and news servers. It does so by performing a series of file system operations such as appends, file reads, directory lookups, creations, and deletions. This benchmark uses little CPU, but is I/O intensive. We configured Postmark to create 1,000 files, between 512–1,045,068 bytes, and perform 5,000 transactions. We chose 1,045,068 bytes as the file size as it was the average inbox size on our campus mail server.

The third benchmark we ran was to copy all the incremental weekly CVS snapshots of the Am-Utils source tree for 10 years onto Versionfs. This simulates the modifications that a user makes to a source tree over a period of time. There were 128 snapshots of Am-Utils, totaling 51,636 files and 609.1MB.

The recover benchmark recovers all the versions of all regular files in the tree created by the copy benchmark. This measures the overhead of accessing a previous version of a file.

5.2.3 Am-Utils Results

Figure 5.1 and Table 5.2 show the performance of Versionfs for an Am-Utils compile with the NUMBER retention policy and 16 versions of each file retained. After compilation, the total space occupied by the Am-Utils directory on Ext3 is 33.3MB.

	Ext3	**Full**	**Compress**	**Sparse**
Elapsed	195.9s	197.0s	201.8s	197.3s
System	43.7s	44.5s	49.4s	44.8s
Space	33.3MB	45.8MB	39.5MB	41.8MB
Overhead over Ext3				
Elapsed	-	0.5%	3.0%	0.6%
System	-	1.8%	13.0%	2.5%

Table 5.2: Am-Utils results.

For FULL, we recorded a 0.5% increase in elapsed time over Ext3, a 1.8% increase in system time over Ext3, and space consumed was 1.37 times that of Ext3. With COMPRESS, the elapsed time increased by 3% over Ext3, the system time increased by 13% over Ext3, and space consumed was 1.18 times that of Ext3. The system time and consequently the elapsed time increase because each version needs to be compressed. With SPARSE, the elapsed time increased by 0.6% over Ext3, the system time increased by 2.5% over Ext3, and the space consumed was 1.25 times that of Ext3. The increase in system time is because there are more write operations than FULL, as

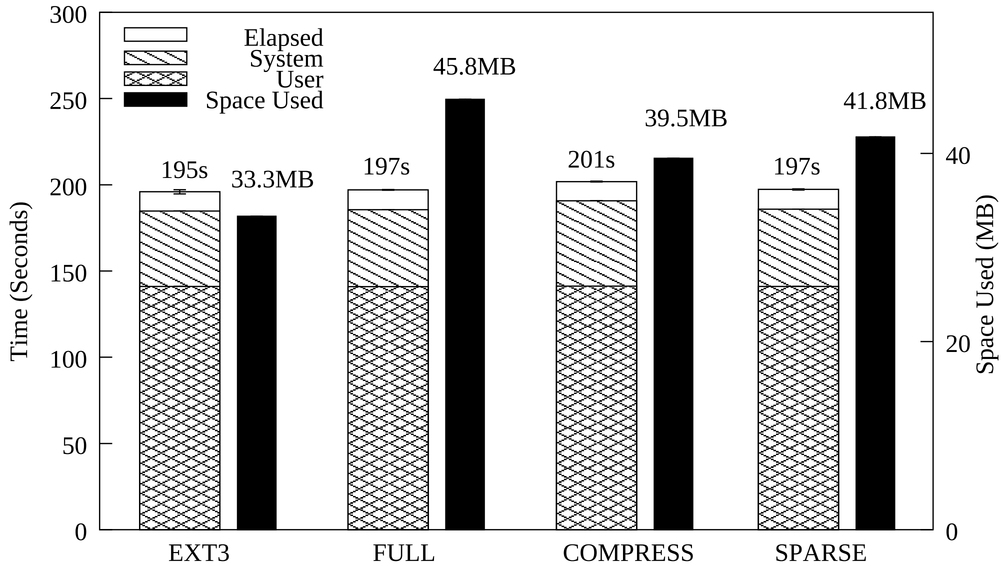

Figure 5.1: Am-Utils Compilation results. Note: benchmark times use the left scale. Space occupied by each configuration at the end of compilation is represented by the thin black bars and use the right scale.

SPARSE has to write both data and meta-data pages; however, SPARSE consumes less disk space. We do not report the Wait times as the variations in them are negligible.

5.2.4 Postmark Results

Figure 5.2 and Table 5.3 show the performance of Versionfs for Postmark with the NUMBER retention policy and 5 versions of each file retained. Postmark deletes all the files at the end of the benchmark, so on Ext3 no space is occupied at the end of the test. Versionfs creates versions, so there will be files left at the end of the benchmark.

For FULL, elapsed time was observed to be 3.26 times, system time 2.13 times and wait time 3.60 times that of Ext3. The increase in the system time is because extra processing has to be done for making versions of files. The increase in the wait time is because additional I/O must be done in copying large files. The overheads are expected since the version files consumed 3.7GB of space at the end of the test.

For COMPRESS, elapsed time was observed to be 11.18 times, system time 33.12 times and wait time 5.02 times that of Ext3. The increase in the system time is due to the large files being compressed while creating the versions. The wait time increases compared to FULL despite having

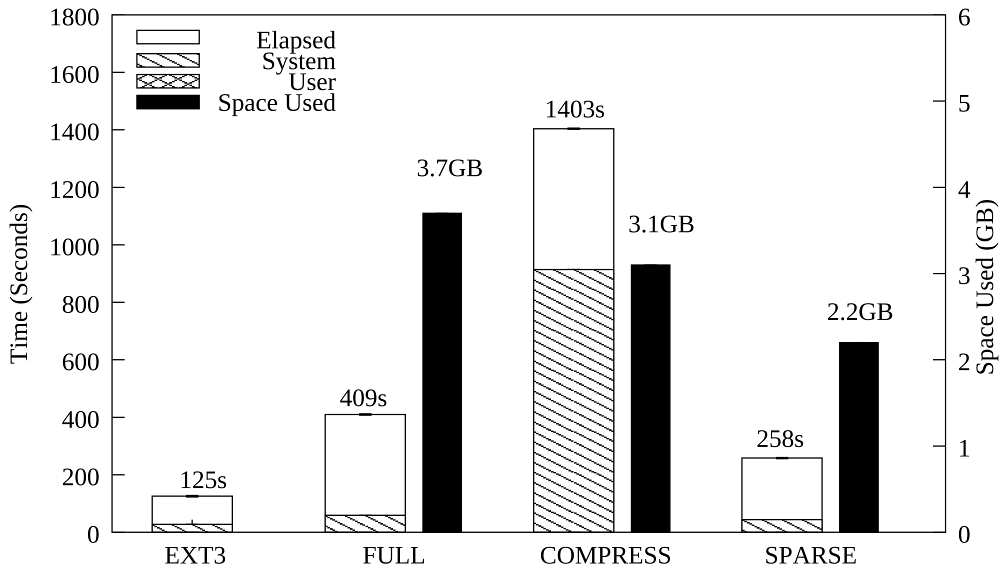

Figure 5.2: Postmark results. Note: benchmark times use the left scale. Space occupied by each configuration at the end of the test is represented by the thin black bars and use the right scale.

	Ext3	**Full**	**Compress**	**Sparse**
Elapsed	125.4s	409.7s	1403.9s	258.0s
System	27.6s	58.8s	914.2s	43.9s
Wait	97.2s	350.1s	488.9s	213.3s
Space	0GB	3.7GB	3.1GB	2.2GB
Overhead over Ext3				
Elapsed	-	3.26 ×	11.18×	2.06×
System	-	2.13 ×	33.12×	1.59×
Wait	-	3.60 ×	5.02×	2.19×

Table 5.3: Postmark results.

to write less data. This is because in FULL mode, unlinks are implemented as a rename at the lower level, whereas COMPRESS has to read in the file and compress it. Since 29% of the operations in our configuration of Postmark are unlinks, this results in a significant overhead. The version files consumed 3.1GB of space at the end of the benchmark.

SPARSE has the best performance both in terms of the space consumed and the elapsed time. This is because all writes in Postmark are appends. In SPARSE, only the page that is changed along with the meta-data is written to disk for versioning, whereas in FULL and COMPRESS, the whole file is written. For SPARSE, elapsed time was 2.06 times, system time 1.59 times, and wait time 2.19 times that of Ext3. The residual version files consumed 2.2GB. The overheads seen in the sparse mode are comparable to CVFS [19], which uses specialized data structures for storing data and organizing versions in a directory.

5.2.5 Am-Utils Copy Results

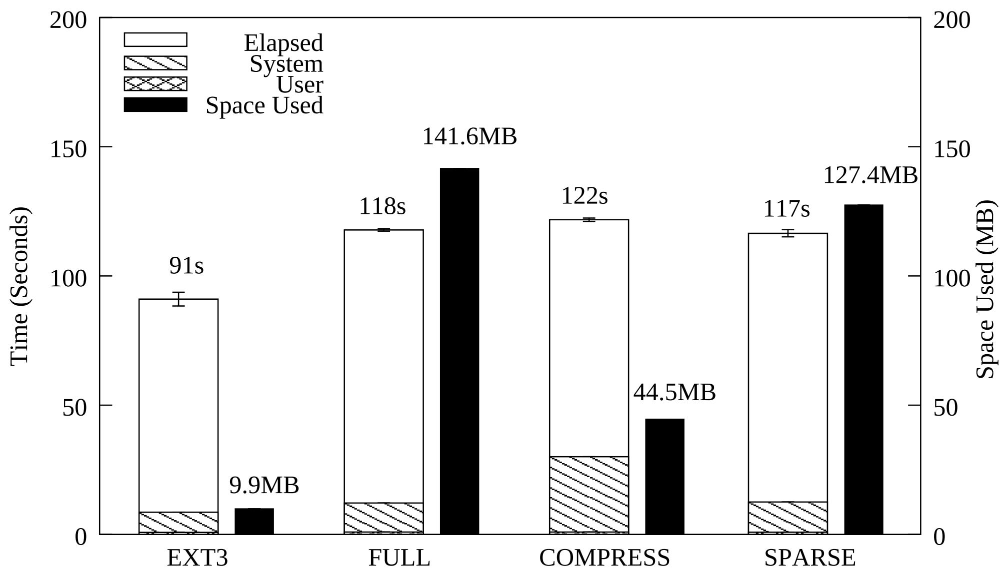

Figure 5.3: Am-Utils Copy Results with NUMBER*=64. Note: benchmark times use the left scale. Space occupied by each configuration at the end of copy is represented by the thin black bars and use the right scale.*

	Ext3	Full	Compress	Sparse
Elapsed	91.0s	117.8s	121.7s	116.5s
System	7.9s	11.3s	29.1s	11.6s
Wait	82.3s	105.7s	91.7s	104.0s
Space	9.9MB	141.6MB	44.5MB	127.4MB
Overhead over Ext3				
Elapsed	-	29.4%	33.7%	28.0%
System	-	41.8%	266.9%	46.5%
Wait	-	28.5%	11.5%	26.5%

Table 5.4: Am-Utils copy results with NUMBER *and 64 versions being retained.*

5.2.5.1 Number

Figure 5.3 and Table 5.4 show the performance of Versionfs for an Am-Utils copy with the NUMBER retention policy and 64 versions of each file retained. We chose 64 versions as it was half the number of versions of Am-Utils snapshots that we had. GNU `cp` opens files for copying with the `O_TRUNC` flag turned on. If the file already exists, it gets truncated and causes a version to be created. To avoid this, we used a modified `cp` that does not open files with O_TRUNC flag but instead truncates the file only if necessary at the end of copying. After copying all the versions, the total space consumed by the Am-Utils directory on Ext3 was 9.9MB.

For FULL, we recorded a 29.4% increase in elapsed time, a 41.8% increase in system time, and a 28.5% increase in wait time over Ext3. FULL consumes 14.35 times the space consumed by Ext3. The system time increases due to two reasons. First, each page is compared and a version is made only if at least one page is different. Second, additional processing must be done in the kernel for making versions. Though the comparison of pages is expensive, without it, FULL consumes 370.7MB compared to 141.6MB that FULL now consumes. This is because many files do not change between one Am-utils snapshot to the other and they would all result in another version without the comparison.

For COMPRESS, we recorded a 33.7% increase in elapsed time, a 266.9% increase in system time, and a 11.5% increase in wait time over Ext3. The increase in the system time is due to version files being compressed. The wait time overhead is the least for the COMPRESS mode as it writes the least amount of data. As all the versioned files are compressed, the space occupied is the least among all the storage modes and it consumes 4.51 times the space consumed by Ext3.

SPARSE has the best performance in terms of the elapsed time. We recorded a 28.0% increase in elapsed time, a 46.5% increase in system time, and a 26.5% increase in the wait time over Ext3. Even though SPARSE writes more data than the compress mode, it performs better as it does not have to compress the data. SPARSE performs better than the normal mode as it has to write only the changed data. SPARSE consumes 12.91 times the space consumed by Ext3.

5.2.5.2 Space

Figure 5.4 and Table 5.5 show the performance of Versionfs for an Am-Utils copy with the SPACE retention policy and each version set having an upper bound of 500KB. We observed that the number of version files and the space occupied by version files decreased. This is because fewer versions of larger files and more versions of smaller files were retained.

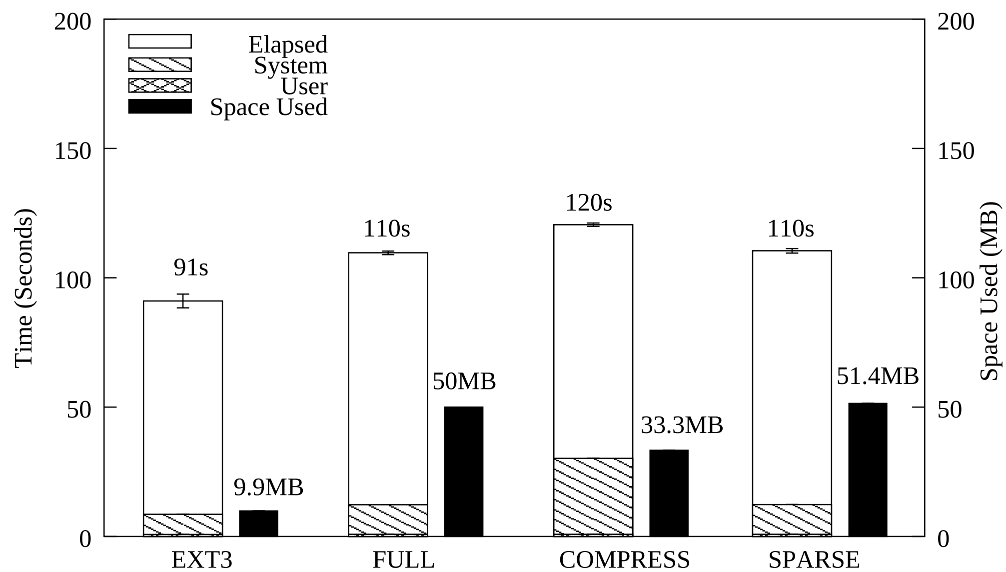

Figure 5.4: Am-Utils Copy Results with SPACE*=500KB. Note: benchmark times use the left scale. Space occupied by each configuration at the end of copy is represented by the thin black bars and use the right scale.*

For FULL, we recorded a 20.4% increase in elapsed time, a 43.7% increase in system time, and an 18.4% increase in the wait time over Ext3. FULL consumes 5.07 times the space consumed by Ext3.

For COMPRESS, we recorded a 32.4% increase in elapsed time, a 268.9% increase in system time, and 9.9% increase in the wait time over Ext3. COMPRESS consumes 3.38 times the space consumed by Ext3.

	Ext3	Full	Compress	Sparse
Elapsed	91.0s	109.6s	120.5s	110.4s
System	7.9s	11.4s	29.3s	11.5s
Wait	82.3s	97.4s	90.4s	98.1s
Space	9.9MB	50.0MB	33.3MB	51.4MB
Overhead over Ext3				
Elapsed	-	20.4%	32.4%	21.3%
System	-	43.7%	268.9%	45.3%
Wait	-	18.4%	9.9%	19.2%

Table 5.5: Am-Utils copy results with SPACE *and 500KB being retained per version set*

SPARSE recorded a 21.3% increase in elapsed time, a 45.3% increase in system time, and a 19.2% increase in wait time over Ext3. SPARSE consumes 5.21 times the space consumed by Ext3. SPARSE takes more space than FULL as it retains more version files than FULL. This is because SPARSE stores only the pages that differ between two versions of a file and hence has smaller version files. We can only discard versions in whole numbers, so SPARSE more closely approaches the 500KB limit per version set, i.e., it packs smaller files better within the limit.

5.2.6 Recover Benchmark Results

Table 5.6 shows results of the recover benchmark. This benchmark recovers all the versions of all files that were created by the copy benchmark. The average times to recover a single file in FULL, COMPRESS, and SPARSE are 8.6ms, 6.4ms, and 9.3ms, respectively. COMPRESS is the fastest as it has to do little I/O and decompression is faster than compression. Sparse mode is the slowest as it has to reconstruct files from multiple versions as described in Section 3.2.3. The recovery time for normal is the time to copy the required version inside of the kernel.

Mode	Time	Files	Avg. Time
Full	27.9s	3224	8.6ms
Compress	20.7s	3224	6.4ms
Sparse	30.1s	3224	9.3ms

Table 5.6: Recover results

Our performance evaluation demonstrates that Versionfs has an overhead for typical user-like workloads of just 1–3%. With an I/O-intensive workload, Versionfs using SPARSE is 2.06 times slower than Ext3, which is comparable with the overhead of other recent versioning systems [19].

For the `cp` benchmark, copy-on-change reduced the space utilization of FULL mode by 61% when compared to copy-on-write. With all storage policies, recovering a single version took less than 10ms.

Chapter 6

Conclusions

The main contribution of this work is that Versionfs allows users to manage their own versions easily and efficiently. Versionfs provides this functionality with less than a 3% overhead for typical user-like workloads. Versionfs allows users to select not only what versions are kept through a combination of retention policies, but also how to store versions through flexible storage policies. Users can select the trade-off between space and performance that best meets their individual needs: full copies, compressed copies, or block deltas. Though users are given control over their versions, the system administrator maintains the ability to enforce minimum and maximum values, and provide sensible defaults for users.

Additionally, through the use of libversionfs, unmodified applications can examine, manipulate, and recover versions. Rather than requiring users to learn separate commands, or ask the system administrator to remount the file system, they can simply run familiar tools to access previous file versions. Without libversionfs, previous versions are completely hidden from users.

Finally, Versionfs goes beyond the simple copy-on-write employed by past systems: we implement copy-on-change. Though at first we expected that the comparison between old and new pages would be too expensive, we found that the increase in system time is more than offset by the reduced I/O and CPU time associated with writing unchanged blocks. When more expensive storage policies are used (e.g., compression), copy-on-change is even more useful.

6.1 Future Work

Many applications operate by opening a file using the `O_TRUNC` option. This behavior wastes resources because data blocks and inode indirect blocks must be freed, only to be immediately reallocated. This behavior also interacts poorly with versioning systems because the existing data is entirely deleted before the new data is written, so versioning systems cannot compare the old data with the new data. Unfortunately, many applications operate in the same way and changing them would require significant effort. We plan to implement *delayed truncation*, so that instead of immediately discarding truncated pages, they are kept until a write occurs and can be compared with the new data. This way we can reduce the number of copy-on-change operations that must occur.

We will extend the system so that users can register their own retention policies.

Users may want to revert back the state of a set of files to the way it was at a particular time. We plan to enhance *libversionfs* to support time-travel access. Rather than remembering what time versions were created, users like to give *tags*, or symbolic names, to sets of files. We plan to store tags and annotations in the version meta-data file.

We will also investigate other storage policies. First, we plan to combine sparse and compressed versions. Preliminary tests indicate that sparse files will compress quite well because there are long runs of zeros. Second, presently sparse files depend on the underlying file system to save space. We plan to create a storage policy that efficiently stores block deltas without any logical holes. Third, we plan to make use of block checksums to store the same block only once, as is done in Venti [15]. Finally, we plan to store only plain-text differences between files. Often when writes occur in the middle of a file, then the data is shifted by several bytes, causing the remaining blocks to be changed. We expect plain-text differences will be space efficient.

Bibliography

[1] B. Berliner and J. Polk. Concurrent Versions System (CVS). `www.cvshome.org`, 2001.

[2] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Francisco, CA, 1992.

[3] Digital Equipment Corporation. *TOPS-20 User's Guide (Version 4)*, January 1980.

[4] D. Ellard, J. Ledlie, and M. Seltzer. The Utility of File Names. Technical Report TR-05-03, Computer Science Group, Harvard University, March 2003.

[5] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, June 2003.

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

[7] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, January 1994.

[8] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[9] D. G. Korn and E. Krell. The 3-D File System. In *Proceedings of the USENIX Summer Conference*, pages 147–156, Summer 1989.

[10] LEGATO. LEGATO NetWorker. `www.legato.com/products/networker`.

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

[12] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleinman, and S. Owara. SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 117–129, 2002.

[13] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[14] Zachary N. J. Peterson and Randal C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadat for a Time-Shifting File System. Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University, 2003. `http://hssl.cs.jhu.edu/papers/peterson-ext3cow03.pdf`.

[15] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies*, pages 89–101, January 2002.

[16] IBM Rational. Rational clearcase. `www.rational.com/products/clearcase/index.jsp`.

[17] W. D. Roome. 3DFS: A Time-Oriented File Server. In *Proc. of the Winter 1992 USENIX Conference*, pages 405–418, San Francisco, California, 1991.

[18] 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 *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, December 1999.

[19] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, March 2003.

[20] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 165–180, October 2000.

[21] VERITAS. VERITAS Backup Exec. `http://www.veritas.com/products/category/ProductDetail.jhtml?productId=%bews`.

[22] VERITAS. VERITAS Flashsnap. `www.veritas.com/van/products/flashsnap.html`.

[23] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, June 2001.