A Versatile and User-Oriented Versioning File System

Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok
Stony Brook University

Appears in the proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)

Abstract

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 solutions 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 and provides a host of user-configurable policies: versioning by users, groups, processes, or file names and extensions; version retention policies and version storage policies. 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 can 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.

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 [2]. CVS allows a group of users to record changes to files in a repository, navigate branches, and recover any version officially recorded in a CVS repository. However, CVS does not work transparently with all applications.

Another form of versioning is backup tools such as Legato's Networker [15] or Veritas's Backup Exec [27] and FlashSnap [28]. Modern backup systems include specialized tools for users to browse a history of file changes and to initiate a recovery of 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.

Another approach is to integrate versioning into the file system [4, 17, 24, 25]. Developing a native versioning file system from scratch is a daunting task and will only work for one file system. Instead, we developed a stackable file system called *Versionfs*. A stackable file system operates at the highest possible layer inside the OS. 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 be flexible for users and administrators, Versionfs supports various *retention* and *storage* policies. Retention policies determine how many versions to keep per file. Storage policies determine how versions are stored. 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 our system under Linux. Our performance evaluation shows that the overheads are not noticeable by users under normal workloads.

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

2 Background

Versioning was provided by some early file systems like the Cedar File System [7] and 3DFS [13]. 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. CVS [2] is a 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.

Snapshotting or check-pointing is another approach for versioning, which is common for backup systems. Periodically a whole or incremental image of a file system is made. The snapshots are available on-line and users can access old versions. Such a system has several drawbacks, the largest one being that changes made between snapshots can not be undone. Snapshotting systems treat all files equally. This is a disadvantage because not all files have the same usage pattern or storage requirements. When space must be recovered, whole snapshots must be purged. Often, managing such snapshot systems requires the intervention of the administrator. Snapshotting systems include AFS [12], Plan-9 [20], WAFL [8], Petal [14], Episode [3], Venti [21], Spiralog [10], a newer 3DFS [22] system, File-Motel [9], and Ext3COW [19]. Finally, programs such as `rsync`, `rdiff`, and `diff` are also used to make efficient incremental backups.

Versioning with copy-on-write is another technique that is used by Tops-20 [4], VMS [16], Elephant File System [24], and CVFS [25]. 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 FFS-like 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 was designed with security in mind. Each individual write or small meta-data change (e.g., atime updates) are versioned. 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.

NTFS version 5, released with Windows 2000, provides *reparse points*. Reparse points allow applications to enhance the capabilities of the file system without requiring any changes to the file system itself [23]. Several backup products, such as Storeactive's LiveBackup [26], are built using this feature. These products are specific to Windows, whereas Versionfs uses stackable templates and can be ported to other OSes easily.

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.

3 Design

We designed Versionfs with 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 common 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*. We implemented Versionfs as a kernel-level file system so that applications need not be modified for accessing the current version. For previous version access, we 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 [30]. 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. Section 3.3 describes retention policies. Section 3.4 describes how previous versions are accessed and manipulated. Section 3.5 describes our version cleaning daemon. Section 3.6 describes file system crash recovery.

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. We restrict the user from directly creating or accessing files with names matching the above pattern. Previous versioning systems, like the Cedar File System, have used a similar naming convention.

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 version set to improve performance. This file allows Versionfs to quickly identify versions and know what name to assign to a new version. 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. The dirty bit that the OS or hardware provides is not sufficient, because it does not distinguish between data being overwritten with the same content or different one.

There are six types of operations that create a version: writes (either through `write` or `mmap` writes), `unlink`, `rmdir`, `rename`, `truncate`, and ownership or permission modifications.

The `write` operations are intercepted by our stackable file system. Versionfs creates a new version if the existing data and the new data differ. Between each open and close, only one version is created. This heuristic approximates one version per save, which is intuitive for users and is similar to Elephant's behavior [24].

The `unlink` system call 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. Translating `unlink` to a `rename` reduces the amount of I/O required for version creation.

The `rmdir` system call is converted into a rename, for example `rmdir foo` renames `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 either versions or version set meta-data files. Deleted directories cannot be accessed unless a user recovers the directory. Directory recovery can be done using the user-level library that we provide (see Section 3.4).

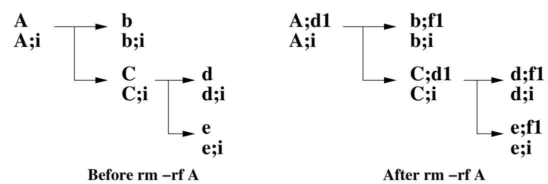

Figure 1: rm -rf on directory A

Figure 1 shows a tree before and after it is removed by `rm -rf`. The `rm` command 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`.

The `rename` system call 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. Whereas we preserve the history of changes to the data in a file, we do not preserve the filename history of a file. This is because we believe that data versioning is more important to users than file-name versioning.

When renaming `foo` to `bar`, if both are regular files, the following three scenarios are possible:

1. **`bar` does not exist:** In this case, we create a version of `foo` before renaming `foo` to `bar`. If both operations succeed, then we create the meta-data file `bar;i`.
2. **`bar` exists:** We first create a version of `bar`. We then create a version of `foo`. Finally, we rename `foo` to `bar`.

3. **`bar` does not exist but `bar;i` exists:** This happens if `bar` has already been deleted and its versions and meta-data files were left behind. In this case, we first create a version for `foo`, then rename `foo` to `bar`. For versioning `bar`, we use the storage policy that was recorded in `bar;i`.

The `rename` system call renames only the head version of a version set. Entire version sets can be renamed using the user-level library we provide (see Section 3.4).

The `truncate` system call must also create a new version. However, when truncating a file `foo` to zero bytes, instead of creating a new version and copying `foo` into the version file, Versionfs renames `foo` to be the version. Versionfs then recreates an empty file `foo`. 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 storage policies: full, compressed, and sparse mode.

Full Mode Full mode makes an entire copy of the file each time a version is created. As can be seen in Figure 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`. Before version 8 is created, its contents are located in `foo`. When the page A2 overwrites the page A1, Versionfs copies the entire head version to the version, `foo;f8`. After the version is created, A2 is written to `foo`, then B1, C2, and D2 are written without any further version creation. This demonstrates that in full mode, once the version is created, there is no additional overhead for read or write. The creation of version 9 is similar to the creation of version 8. The first write overwrites the contents of page A2 with the same contents. Versionfs does not create a version as the two pages are the same. When page B2 overwrites page B1, the contents of `foo` are copied to `foo;f9`. Further writes directly modify `foo`. Pages C2, D3, and E1 are directly written to the head version. Version 10 is created in the same way. Writing A2 and B2 do not create a new version. Writing C3 over C2 will create the version `foo;f10` and the head file is copied into `foo;f10`. Finally, the file is truncated. Because a version has already been created in the same session, a new version is not created.

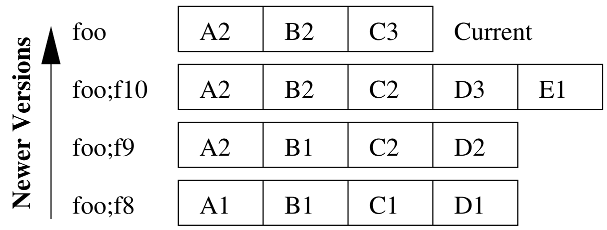

Figure 2: Full versioning. Each version is stored as a complete copy and each rectangle represents one page.

Compress Mode Compress mode is the same as full mode, except that the copies of the file are compressed. 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.

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 intermediate data pages, but does leave logical holes. These holes allow Versionfs to backup changed pages on future writes without costly data-shifting operations [29].

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;s`N. Versionfs reconstructs the file one page at a time. If a page is missing from `foo;s`N, 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.

Figure 3 shows the contents of `foo` when no versions exist. A meta-data file, `foo;i`, which contains the next

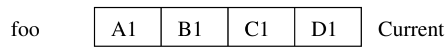

Figure 3: Sparse versioning. Only `foo` exists.

version number, also exists. Figure 4 shows the version set after applying the same sequence of operations as in Figure 2, but in sparse mode.

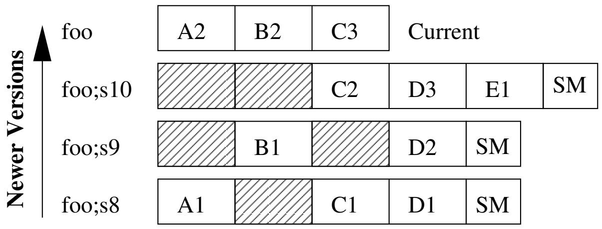

Figure 4: Sparse versioning. Each version stores only block deltas, and each rectangle represents one page. Rectangles with hatch patterns are sparse. Sparse meta-data is represented by the rectangle with "SM" inside.

Versionfs creates `foo;s8` when `write` tries to overwrite page A1 with A2. Versionfs first allocates a new disk block for `foo;s8`, writes A1 to the new block, updates the sparse bitmap and then overwrites A1 with A2 in `foo`. This strategy helps preserve sequential read performance for multi-block files. The other data blocks are not copied to `foo;s8` yet and `foo;s8` remains open. Next, `write` overwrites page B1 with the same data. Versionfs does not write the block to the sparse file because data has not changed. Next, C2 overwrites C1 and Versionfs first writes C1 to the sparse file and then writes C2 to the head version. Versionfs also updates the sparse meta-data bitmap. Page D is written in the same way as page C. The creation of version 9 is similar to version 8. The last version in this sequence is version 10. The pages A2, B2, and C3 are written to the head version. Only C3 differs from the previous contents, so Versionfs writes only C2 to the version file, `foo;s10`. Next, the file is truncated to 12KB, so D3 and E1 are copied into `foo;s10`. The resulting version is shown in Figure 4.

3.3 Retention Policies

We have developed three version retention policies. Our retention policies, as well as Elephant's [24] retention policies, determine how many versions must be retained for a file. However, we provide policies that are different from the ones provided by Elephant. We support the following three retention policies:

Number: The user can set the maximum and minimum number of versions in a version set. This policy is attractive because 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 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 violates 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 is useful when a combination of two policies is 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 [5]. This type of policy could be used to exclude large multimedia files or regenerable files such as `.o` files. This can also be used to prevent applications from creating excessive versions of unwanted files. For example, excluding ˜ from versioning will prevent `emacs` from creating multiple versions of ˜ files.

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. Users can customize these policies. The administrator can set the minimum and maximum values for each policy. This is useful to ensure that users do not abuse the system. In case of conflicts, administrator-defined values override user-defined values. In case of conflicts between two retention policies specified by a user, the most restrictive policy takes precedence.

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 off 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* to convert standard system call wrappers into Versionfs `ioctl`s. The libversionfs library 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, libversionfs 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 libversionfs invokes the desired function in terms of Versionfs `ioctl`s; otherwise the standard library wrapper is used. The `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.

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. This `ioctl` also returns the same information as `stat` for each version.

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 gives application programmers a great deal of flexibility. Using an appropriate descriptor they can recover a version, append the version to an existing file, or stream 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 libversionfs to `open` a version file. To preserve the version history integrity, Version files can be opened for reading only. The libversionfs library 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 that can be used to read the contents of a version.

Open a Raw Version File Opening a raw version returns a file descriptor to an underlying version file. Users are not allowed to modify raw 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 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`, libversionfs runs `readdir` on the current version of the raw directory so that deleted versions are returned to user space. The contents of the underlying directory are then interpreted by libversionfs to present a consistent view to user space. Deleted directories cannot be opened through standard VFS calls, therefore we use the raw open `ioctl` to access them as well.

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 manipulation `ioctl`s. The system administrator can disable these `ioctl`s so that previous versions are not modified.

3.4.1 Operational Scenario

All versions of files are exposed by libversionfs. For example, version 8 of `foo` is presented as `foo;8` regardless of the underlying storage policy. Users 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 libversionfs is as follows. Normally users see only the head version, `foo`.

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

Next, users set an LD_PRELOAD to see all versions.

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

After using libversionfs 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, libversionfs presents a consistent interface. The second version of `foo` is named `foo;2`. There are no modifications required to standard applications.

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

If users want 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
```

libversionfs can also be used to modify an entire version set. For example, the standard `mv` command can be used to rename every version in the version set.

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

3.5 Version Cleaner and Converter

Using the version-set stat and open raw `ioctls` 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).

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 take 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 long period of time.

3.6 Crash Recovery

In the event of a crash, 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. Versionfs, however, depends on the lower level file system to ensure consistency of files and file names. We provide a high-level file system checker (similar to `fsck`) to reconstruct damaged or corrupt version meta-data files.

4 Implementation

We implemented our system on Linux 2.4.20 starting from a stackable file system template [30]. Versionfs is 12,476 lines of code. Out of this, 2,844 lines were for the various `ioctls` that we implemented to recover, access, and modify versions. Excluding the code for the `ioctls`, 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.

The stackable file system templates we used cache pages both on the upper-level and lower-level file system. We take advantage of the 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` 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 is reduced significantly.

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 several 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 WAFL [8], Elephant [24], CVFS [25], and Versionfs. We selected these because they version files without any user intervention. We do not include some of the older systems like Tops-20 [4] and VMS [16] as they do 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. We chose WAFL as it is a recent representative of snapshotting systems. We do not include Venti [21] as it provides a framework that can be used for versioning rather than being a versioning system itself.

	Feature	WAFL	Elephant	CVFS	Versionfs
1	File system implementation method	Disk based	Disk based	Disk based [a]	Stackable
2	Copy-on-Change				✔
3	Comprehensive versioning (data, meta-data, etc.)			✔[b]	
4	Transparent support for compressed versions				✔
5	Landmark retention policy		✔		
6	Number based retention policy		[c]		✔
7	Time based retention policy		✔		✔
8	Space based retention policy				✔
9	Unmodified applications can access previous versions	✔			✔
10	Per-user extension inclusion/exclusion list to version				✔
11	Administrator can override user policies				✔
12	Allows users to register their own reclamation policies		✔		
13	Version tagging		✔		

Table 1: Feature comparison. A check mark indicates that the feature is supported, otherwise it is not.
[a] Log structured disk-based file system with an NFS interface.
[b] Security audit quality versioning.
[c] Elephant supports "keep all" and "keep one" policies.

We identified the following thirteen features and have summarized them in Table 1:

1. **File system implementation method:** Versionfs is implemented as a stackable file system, so it can be used with any file system. WAFL, Elephant, and CVFS are disk-based file systems and cannot be used in conjunction with any other file system. CVFS uses log structured disk layout and exposes an NFS interface for accessing the device. The advantage of stacking is that the underlying file system can be chosen based on the users' needs. For example, if the administrator expects a lot of small files to be created in one directory, the user can stack on top of a hash-tree based or tail-merging file system to improve performance and disk usage.
2. **Copy-on-Change:** Versionfs takes advantage of the double-buffering used by stackable templates. On writes, Versionfs compares each byte of the new data with the old data and makes a version only if there are changes. This is advantageous as several applications (e.g., most text editors) rewrite the same data to disk. CVFS and Elephant use copy-on-write for creating new versions, creating versions when there actually is no change.
3. **Comprehensive versioning:** Comprehensive versioning creates a new version on every operation that modifies the file or its attributes. This is particularly useful for security purposes. Only CVFS supports this feature.
4. **Transparent support for compressed versions:** Versionfs supports a policy that allows versions to be stored on disk in a compressed format. In Elephant, a user-land cleaner can compress old files. In Versionfs, users can access the data in compressed files directly by using libversionfs. In Elephant users cannot access the version files directly.
5. **Landmark retention policy:** This policy retains only important or landmark versions in a file's history. Only Elephant supports this policy.
6. **Number based retention policy:** This policy lets the user set upper and lower bounds on the number of versions retained for a file. Versionfs supports this policy. Elephant supports "keep one" policy that retains 0 versions and "keep all" policy that retains infinite versions.
7. **Time based retention policy:** This policy retains versions long enough to recover from errors but versions are reclaimed after a configured period of time. Versionfs supports this policy. Elephant's "keep safe" policy is functionally similar to Versionfs's time based policy.
8. **Space based retention policy:** This policy limits the space used by a file's versions. Only Versionfs supports this policy.
9. **Unmodified application access to previous versions:** Versioning is only truly transparent to the user if previous versions can be accessed without making modifications to user-level applications like `ls`, `cat`, etc. This is possible in Versionfs (through libversionfs) and in WAFL. Elephant and CVFS need modified tools to access previous versions.
10. **Per-user extension inclusion and exclusion lists 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. Versionfs allows users to specify a list of extensions to be excluded

from versioning. Elephant allows groups of files to have the same retention policies, but does not allow explicit extension lists.

11. **Administrator can override user policies:** Providing flexibility to users means users could misconfigure the policies. It is important that administrators can override or set bounds for the user policies. Versionfs allows administrators to set an upper bound and lower bound on the space, time, and number of versions retained.
12. **Allows users to register their own reclamation policies:** Users might prefer policies other than the system default. Elephant is the only file system that allows users to set up custom reclamation policies.
13. **Version tagging:** Users want to mark the state of a set of files with a common name so that they can revert back to that state in the future. Only Elephant supports tagging.

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 two 18GB 15,000 RPM Maxtor Atlas hard drives configured as a 32GB software Raid 0 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 [6]. We recorded elapsed, system, and user times, and the amount of disk space utilized for all tests. We display elapsed times (in seconds) on top of the time bars in all our graphs. 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. It is computed as the difference between the elapsed time and system+user times. 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 times 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. 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 evaluation:

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

We used the following retention policies:

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

For all benchmarks, one storage and one retention configuration were 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 [18]. 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 and retention policies that we support. However, 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. For this benchmark, 25% of the operations are writes, 22% are `lseek` operations, 20.5% are reads, 10% are `open` operations, 10% are `close` operations, and the remaining operations are a mix of `readdir`, `lookup`, etc.

The second workload we chose was Postmark [11]. Postmark simulates the operation of electronic mail servers. It does so by performing a series of file system operations such as appends, file reads, creations, and deletions. This benchmark uses little CPU, but is I/O intensive. We configured Postmark to create 5,000 files, between 512–1,045,068 bytes, and perform 20,000 transactions. For this configuration, 58% of the operations are writes, 37.3% are reads and the remaining are a mix of operations like `open`, `flush`, etc. (We use Tracefs [1] to measure the exact operation mix in the Am-Utils and Postmark benchmarks.) We chose 1,045,068 bytes as the file size as it was the average inbox size on our large 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. We end the section with statistics to compare Copy-on-write with Copy-on-change.

5.2.3 Am-Utils Results

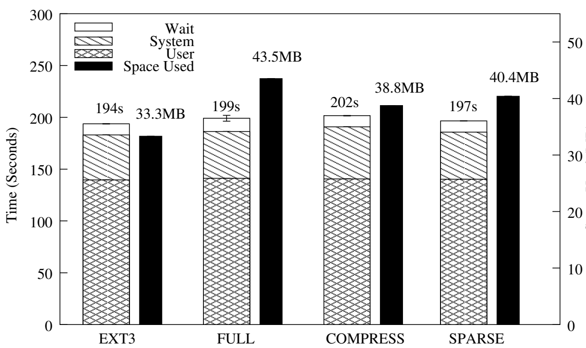

Figure 5: Am-Utils Compilation results. Note: benchmark times use the left scale. Space occupied by each configuration at the end of compilation is represented by the black bars and use the right scale.

	Ext3	Full	Compress	Sparse
Elapsed	193.7s	199.1s	201.6s	196.6s
System	43.4s	45.2s	50.3s	45.4s
Wait	10.8s	12.7s	10.7s	10.8s
Space	33.3MB	43.5MB	38.8MB	40.4MB
Overhead over Ext3				
Elapsed	-	3%	4%	1%
System	-	4%	16%	5%
Wait	-	18%	-1%	0%

Table 2: Am-Utils results.

Figure 5 and Table 2 show the performance of Versionfs for an Am-Utils compile with the NUMBER retention policy and five versions of each file retained. During the benchmark, all but 15 of the files that are created have less than five versions and the remaining files have between 11 and 1,631 versions created. Choosing five versions as the limit ensures that some of the retention polices are applied. After compilation, the total space occupied by the Am-Utils directory on Ext3 is 33.3MB.

For FULL, we recorded a 3% increase in elapsed time, a 4% increase in system time, and an 18% increase in wait time over Ext3. The space consumed was 1.31 times that of Ext3. The system time increases as each page is checked for changes before being written to disk. The wait time increases due to extra data that must be written. With COMPRESS, the elapsed time increased by 4%, the system time increased by 16%, and the wait time decreased by 1% over Ext3. The space consumed was 1.17 times that of Ext3. The system time and consequently the elapsed time increase because each version needs to be compressed. COMPRESS has the least wait time, but this gain is offset by the increase in the system time. With SPARSE, the elapsed time increased by 1% and the system time increased by 5% over Ext3. The wait time was the same as Ext3. The space consumed was 1.21 times that of Ext3. SPARSE consumes less disk space than FULL and hence has smaller wait and elapsed time overheads.

5.2.4 Postmark Results

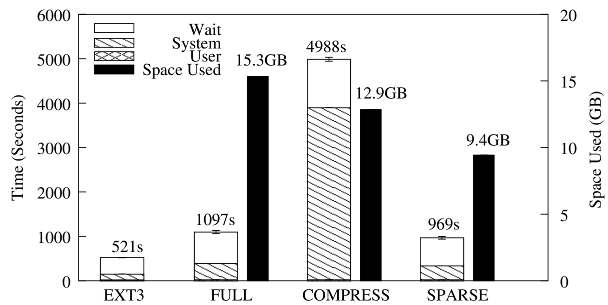

Figure 6: Postmark results. Note: benchmark times use the left scale. Space occupied by each configuration at the end of the test is represented by the black bars and use the right scale.

	Ext3	Full	Compress	Sparse
Elapsed	521s	1097s	4988s	969s
System	128s	368s	3873s	313s
Wait	373s	708s	1093s	634s
Space	0GB	15.34GB	12.85GB	9.43GB
Overhead over Ext3				
Elapsed	-	2.1 ×	9.6×	1.9×
System	-	2.9 ×	30.3×	2.4×
Wait	-	1.9 ×	2.9×	1.7×

Table 3: Postmark results.

Figure 6 and Table 3 show the performance of Versionfs for Postmark with the NUMBER retention policy and nine versions of each file retained. We chose to retain nine versions because eight is the maximum number of versions created for any file and we wanted to retain all the versions of the files. 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 2.1 times, system time 2.9 times, and wait time 1.9 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 15.34GB of space at the end of the test.

For COMPRESS, elapsed time was observed to be 9.6 times, system time 30.3 times and wait time 2.9 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 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. The version files consumed 12.85GB 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 1.9 times, system time 2.4 times, and wait time 1.7 times that of Ext3. The residual version files consumed 9.43GB. The 9.43GB space consumed by SPARSE is the least amount of space consumed for the Postmark benchmark. For a similar Postmark configuration, CVFS, running with a 320MB cache, consumes only 1.3GB, because it uses specialized data structures to store the version meta-data and data. Versionfs cannot optimize the on-disk structures as it is a stackable file system and has no control over the lower-level file system.

9.43GB is more than 9 times the amount of RAM in our system, so these results factor in the effects of double buffering and ensure that our test generated substantial I/O. This benchmark also factors in the cost of `readdir` in large directories that contain versioned files. At the end of this benchmark, 40,053 files are left behind, including versions and meta-data files.

5.2.5 Am-Utils Copy Results

Number Retention Policy Figure 7 and Table 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 is median of the number of versions of Am-Utils snapshots we had. Choosing 64 versions also ensures that the effects of retention policies will also be factored into the results. 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 Am-Utils directory on Ext3 consumes 9.9MB.

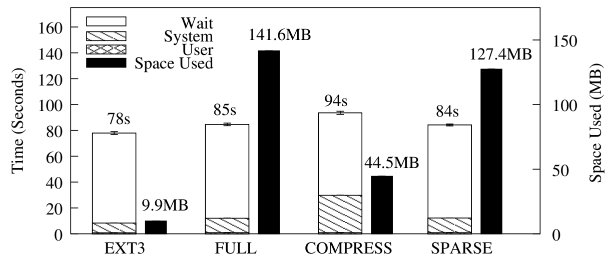

Figure 7: Am-Utils copy results with NUMBER *and 64 versions being retained. Note: benchmark times use the left scale. Space occupied by each configuration at the end of copy is represented by the black bars and use the right scale.*

	Ext3	Full	Compress	Sparse
Elapsed	78.0s	84.6s	93.6s	84.2s
System	7.5s	11.2s	28.9s	11.3s
Wait	69.6s	72.6s	63.8s	72.0s
Space	9.9MB	141.6MB	44.5MB	127.4MB
Overhead over Ext3				
Elapsed	-	8%	20%	8%
System	-	49%	285%	51%
Wait	-	4%	-8%	3%

Table 4: Am-Utils copy results with NUMBER *and 64 versions being retained.*

For FULL, we recorded an 8% increase in elapsed time, a 49% increase in system time, and a 4% 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.

For COMPRESS, we recorded a 20% increase in elapsed time, a 285% increase in system time, and an 8% decrease in wait time over Ext3. Copy-on-change combined with data compression results in wait time less than even Ext3. This is offset by the increase in the system time due to the compression of version files. 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.

For SPARSE, we recorded an 8% increase in elapsed time, a 51% increase in system time, and a 3% increase in the wait time over Ext3. Even though SPARSE writes more data than COMPRESS, SPARSE performs better as it does not have to compress the data. The performance of SPARSE is similar to FULL mode since even a small change at the beginning of the file results in the whole file being written. SPARSE consumes 12.91 times the space consumed by Ext3.

Space Retention Policy Figure 8 and Table 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 140KB. We chose 140KB as it is median of the product of the average number of versions per file and the average version file size when all versions are retained. We observed that the number of version files increased for SPARSE and COMPRESS and decreased for FULL. The space occupied by version files decreased. This is because fewer versions of larger files and more versions of smaller files were retained.

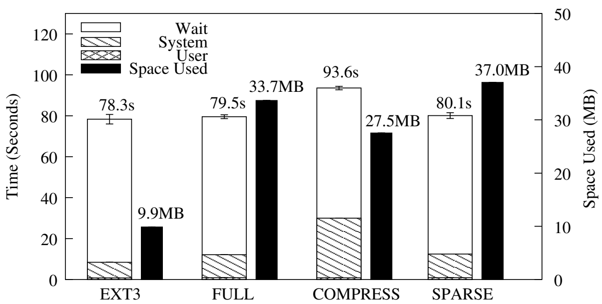

Figure 8: Am-Utils copy results with SPACE *and 140KB being retained per version set. Note: benchmark times use the left scale. Space occupied by each configuration at the end of copy is represented by the black bars and use the right scale.*

	Ext3	Full	Compress	Sparse
Elapsed	78.3s	79.5s	93.6s	80.1s
System	7.7s	11.3s	29.2s	11.6s
Wait	69.9s	67.4s	63.6s	67.6s
Space	9.9MB	33.7MB	27.5MB	37.0MB
Overhead over Ext3				
Elapsed	-	2%	20%	2%
System	-	47%	279%	51%
Wait	-	-4%	-9%	-3%

Table 5: Am-Utils copy results with SPACE *and 140KB being retained per version set.*

For FULL, we recorded a 2% increase in elapsed time, a 47% increase in system time, and an 4% decrease in the wait time compared to Ext3. FULL consumes 3.40 times the space consumed by Ext3.

For COMPRESS, we recorded a 20% increase in elapsed time, a 279% increase in system time, and 9% decrease in the wait time over Ext3. COMPRESS consumes 2.78 times the space consumed by Ext3.

SPARSE has a 2% increase in elapsed time, a 51% increase in system time, and a 3% decrease in wait time over Ext3. SPARSE consumes 3.74 times the space consumed by Ext3. SPARSE takes more space than FULL as it retains more version files. This is because SPARSE stores only the pages that differ between two versions of a file and hence has smaller version files. Consequently, SPARSE packs more files and approaches the 140KB limit per version set more closely.

For all the configurations in this benchmark, we observed that Versionfs had smaller wait times than Ext3. This is because Versionfs can take advantage of Copy-on-change and does not have to write a page if it has not changed. Ext3, however, has to write a page even if it is the same data being overwritten. The system time increases for all configurations as a combined effect of increased `lookup` times and copy-on-change comparisons. System time is the most for COMPRESS as the system has to compress data in addition to the other overheads. Wait time is the least for COMPRESS as the least amount of data is written in this configuration.

5.2.6 Recover Benchmark Results

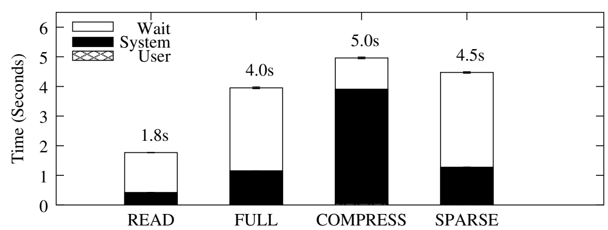

Figure 9: Recover results. READ *is the time to read all files if they were stored as regular files.* FULL, COMPRESS, *and* SPARSE *are the times to recover all versions of all files in the corresponding modes.*

Figure 9 shows the time required to recover all the versions of all files (3,203 versions) that were created by the copy benchmark and the time to read all the versions if they are stored as regular files. The average times to recover a single file in FULL, COMPRESS, SPARSE, and READ were 1.2ms, 1.5ms, 1.4ms, and 0.53ms respectively. The recover times for FULL, COMPRESS, and SPARSE were 2.2, 2.8, and 2.5 times that required to READ, respectively. FULL was the fastest as the recovery time in FULL is the time to copy the required version inside of the kernel. The wait time for FULL increases as the data has to be read from the version file and then written into the recover file. COMPRESS was the slowest as it has to decompress each version. The amount of I/O in COMPRESS is the least as it has to read the least amount of data. However, the time gained in I/O is lost in the system time used for decompressing data. SPARSE mode has the most amount of I/O as it has to reconstruct files from multiple versions as described in Section 3.2.

Benchmark	FULL-COW	FULL-COC	COMPRESS-COW	COMPRESS-COC	SPARSE-COW	SPARSE-COC
Am-Utils copy	370.7MB	141.6MB	168.4MB	44.5MB	370.9MB	127.4MB
Am-Utils compile	43.5MB	43.5MB	38.8MB	38.8MB	41.63MB	40.4MB

Table 6: Copy-on-write (COW) vs. Copy-on-change (COC).

5.2.7 Copy-on-Write vs. Copy-on-Change

Table 6 shows the space consumed by all the storage policies for the Am-Utils copy and Am-Utils compile benchmarks with Copy-on-write and with Copy-on-change. In the Am-Utils copy benchmark, there were considerable savings ranging from 73.6% for COMPRESS to 61.8% for FULL. The savings were good as users generally tend to make minor modifications to the files that copy-on-change can take advantage of.

For the Am-Utils compile benchmark, there is a 2.9% savings in space utilization with copy-on-change in the SPARSE mode and no savings in the FULL and COMPRESS modes. This is because there are more changes to files in Am-Utils compile. Only SPARSE can take advantage of copy-on-change in Am-Utils compile as SPARSE works at a page granularity.

In summary, our performance evaluation demonstrates that Versionfs has an overhead for typical workloads of just 1–4%. With an I/O-intensive workload, Versionfs using SPARSE is 1.9 times slower than Ext3. With all storage policies, recovering a single version takes less than 2ms. Copy-on-change, depending on the load, can reduce disk usage and I/O considerably.

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 no more than 4% overhead for typical user-like workloads. Versionfs allows users to select both what versions are kept and how they are stored through retention policies and storage policies, respectively. Users can select the trade-off between space and performance that best meets their individual needs: full copies, compressed copies, or block deltas. Although users can control their versions, the administrator can enforce minimum and maximum values, and provide users sensible defaults.

Additionally, through the use of libversionfs, unmodified applications can examine, manipulate, and recover versions. Users can simply run familiar tools to access previous file versions, rather than requiring users to learn separate commands, or ask the system administrator to remount a file system. 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. As the existing data is entirely deleted before the new data is written, versioning systems cannot use copy-on-change. 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 can reduce the number of I/O operations that 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 [21]. 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.

7 Acknowledgments

We thank the FAST reviewers for the valuable feedback they provided, as well as our shepherd, Fay Chang. We also thank Greg Ganger and Craig Soules for assisting with extra CVFS benchmarks. 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.

References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, San Francisco, CA, March/April 2004.

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

[9] A.G. Hume. The File Motel – An Incremental Backup System for Unix. In *Summer USENIX Conference Proceedings*, pages 61–72, June 1988.

[10] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, 8(2):5–14, 1996.

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

[12] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25, Asilomar Conference Center, Pacific Grove, CA, October 1991. ACM Press.

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

[14] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, pages 84–92, Cambridge, MA, 1996.

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

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

[17] 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.

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

[19] Z. N. J. Peterson and R. 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`.

[20] S. Quinlan. A Cached WORM File System. *Software – Practice and Experience*, 21(12):1289–1299, 1991.

[21] 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.

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

[23] M. Russinovich. Inside Win2K NTFS, Part 1. `www.winnetmag.com/Articles/ArticleID/15719/pg/2/2.html`, November 2000.

[24] 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.

[25] 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.

[26] Storactive. Storactive LiveBackup. `www.storactive.com/solutions/liveBackup`.

[27] VERITAS. VERITAS Backup Exec. `www.veritas.com/products/category/ProductDetail.jhtml?productId=bews`.

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

[29] 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.

[30] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.