Elephant: The File System that Never Forgets

Douglas J. Santry, Michael J. Feeley, Norman C. Hutchinson
Department of Computer Science
University of British Columbia
Vancouver, Canada
{*dsantry,feeley,norm*}*@cs.ubc.ca*

Alistair C. Veitch
Hewlett Packard Laboratories
Palo Alto, California
aveitch@hlp.hp.com

Abstract

Modern file systems associate the deletion of a file with the release of the storage associated with that file, and file writes with the irrevocable change of file contents. We propose that this model of file system behavior is a relic of the past, when disk storage was a scarce resource. We believe that the correct model should ensure that all user actions are revocable. Deleting a file should change only the name space and file writes should overwrite no old data. The file system, not the user, should control storage allocation using a combination of user specified policies and information gleaned from file-edit histories to determine which old versions of a file to retain and for how long.

This paper presents the Elephant file system, which provides users with a new contract: Elephant will automatically retain all important versions of the users files. Users name previous file versions by combining a traditional pathname with a time when the desired version of a file or directory existed. Elephant manages storage at the granularity of a file or groups of files using user-specified retention policies. This approach contrasts with checkpointing file systems such as Plan-9, AFS, and WAFL, that periodically generate efficient checkpoints of entire file systems and thus restrict retention to be guided by a single policy for all files within that file system. We also report on the Elephant prototype, which is implemented as a new Virtual File System in the FreeBSD kernel.

1. Introduction

Disks are becoming ever cheaper and larger. Human productivity, however, remains constant. This affords system designers an opportunity to re-examine the way we use disk stores. In particular, the previous model of user-controlled storage allocation may no longer be valid.

*This work is supported by the Natural Sciences and Engineering Research Council of Canada.

The current model of most file systems is that they store the most recent version of a file. Users explicitly manage files by deleting them when they are no longer needed or when necessary to make room for new files when storage fills. If a user wants to maintain older versions of files she must explicitly make and maintain multiple copies.

Today, information is valuable and storage is cheap. It thus makes sense for the file system to use some of this cheap storage to ensure that valuable files are never lost due to the failure of a user to make a copy (or make the right copy) before modifying them, or because of the accidental or uninformed deletion of a file that is in fact valuable.

If a file is mistakenly removed or overwritten, valuable data can be lost and in some real sense the file system will have failed the user. As a result, the savvy user tends to be conservative, making many copies of data and avoiding deletes whenever possible. Many researchers have observed this problem and attempted to solve it. Section 2 summarizes this related work.

This paper proposes a new file system model, called Elephant, in which old versions of files are automatically retained and storage is managed by the file system using file-grain retention policies specified by the user. The goal of Elephant is to present users with a file system that retains important old versions of all of their files. User actions such as delete and file write are thus easily revocable by rolling back the file system, a directory, or an individual file to an earlier point in time. Section 3 describes the design of Elephant and Section 4 describes our prototype implementation.

2. Related Work

The goal of keeping multiple versions of data automatically, compactly, and in an organized way is reminiscent of software version control systems [8, 12]. These systems are implemented by application programs running on top of a traditional file system. Users checkout a version from a version-controlled repository, modify a local copy of that version in the file system, and then return the modified version to the repository, which compresses it with respect to older versions. In essence, the goal of Elephant is to extend this idea to all clients of the file system by moving the versioning semantics into the file system, which continues to be accessed using the traditional file system interface of open, close, read, write, etc, and thus freeing users from the details of version management.

While traditional Unix file systems are single-version explicit-delete as we have described, other file system models have been explored in both research and commercial systems. The idea of versioned files was first proposed for the Cedar file system from Xerox PARC [10]. In Cedar, files were immutable; writing to a file produced a new version of the file and file names included a version number (e.g., filename!10). A similar idea was found in the RSX, VMS [2], and TOPS-10/-20 [6] operating systems from Digital.

The approach taken by these systems has two key limitations. First, the maximum number of file versions retained by the system was assigned as a per file parameter; when this threshold was reached, the oldest version was deleted. However, the deletion of the oldest version is a poor heuristic for deciding which files are valuable. Interesting versions of files may be discarded while undesirable or less interesting versions still exist. Second, versioning did not apply to directories. Operations such as renaming a file, creating or destroying a directory, or, in some cases, deleting a file, were thus not revocable.

Several recent file systems have taken a different approach to versioning. In systems such as AFS [5], Plan-9 [7], and WAFL [4] an efficient checkpoint of an entire file system can be created to facilitate backup or to provide users with some protection from accidental deletes and overwrites. A checkpoint is typically created and maintained in a copy-on-write fashion in parallel with the active file system. The old version thus represents a consistent snapshot of the file system sufficient for creating a consistent backup while the file system remains available for modification by users. The snapshot also allows users to easily retrieve an older version of a file.

These systems differ in how frequently checkpoints are taken and in how many checkpoints are retained. In AFS and Plan-9, checkpoints are typically performed daily. In WAFL they can be performed as frequently as every few hours. Plan-9 retains all checkpoints, WAFL keeps the last 32, and AFS keeps only the most recent checkpoint.

Checkpointing file systems have two major limitations. First, checkpoints apply to all files equally, but files have different usage patterns and retention requirements. While it is not feasible to retain every version of every file, it may be important to keep every version of some files. Unfortunately, this dilemma cannot be solved using a file system-grain approach to checkpointing. Elephant addresses this limitation using file-grain retention policies that can be specified by the user. Second, changes that occur between checkpoints cannot be rolled back. For instance, users of daily-checkpointing systems such as Plan-9 or AFS are as as vulnerable as UFS users to losing all their morning's work in the afternoon, due to an inadvertent file deletion or overwrite.

3. Elephant

In Elephant, all user operations are reversible. Deleting a file does not release its storage and file writes are handled in a copy-on-write fashion, creating a new version of a file block each time it is written. Elephant's update handling is thus similar to the Log Structured File System[4, 9, 11], though meta data handling and log cleaning are fundamentally different as discussed below.

File versions are indexed by the time they were created and versioning is extended to directories as well as files. When naming a file or directory, a user can optionally specify a date and time as part of the name. The system resolves this name-time pair to locate the version that existed at the specified time. By rolling a directory back in time, for example, the user will see the directory exactly as it existed at the that earlier time, including any files that the user subsequently deleted. Delete thus becomes a name space management operation; every delete can be undone.

While modern systems have vast storage capacity, this storage is still finite. It is thus still necessary to reclaim storage. In Elephant, users can specify data retention policies on a per-file or per-file-group basis, in a fashion similar to access-protection information in a traditional UNIX file system. Periodically, a file system cleaner examines the file system and uses these policies to decide when and which disk blocks to reclaim, compress, or move to tertiary storage. A variety of policies are possible and necessary to handle the various different types of files stored in the file system.

3.1. Understanding What to Delete and When

Key to the design of Elephant's version retention policies is an understanding of how users use the file system to store various types of files and how the increasing capacity of disk storage impacts on the decisions they make. This section summarizes our observations taken from several UNIX file system traces [3].

Ironically, as storage becomes larger, it becomes more difficult for users to manage files. When storage is fairly constrained, users are required to frequently assess the files that they are maintaining, and delete those that are no longer necessary. Typically, the ability of the user to make this assessment effectively deteriorates over time. After a few weeks or months, the user is unlikely to remember why certain versions are being maintained. While there may be value in maintaining these old versions, it becomes more and more difficult to make sense of them.

Leaving the user to create backup copies of files also requires that the user anticipate that she wants to duplicate a set of files before making a change. Again, this either leads to making excessive copies or sometimes to a situation where a user has just made a change they would like to reverse, but didn't make a copy before making the change, and so is unable to retrieve the old data.

Lastly, as storage space is finite, it will eventually become necessary to delete something. What strategy should the user employ? Unless they have carefully remembered what version is what, their probable strategy is to delete the oldest versions. This, however, is often the wrong strategy because a version history typically contains certain landmark versions surrounded by other versions whose time frame of interest is much shorter. By landmark, we simply mean a distinguished version of a file.

The right strategy to employ when freeing disk resources is to maintain the landmark versions and delete the other versions. Unfortunately, the user may have no good way to tell which old version is important. This process is exacerbated by the fact that these decisions are often made under a time crunch when a disk is full or nearly full.

It is often possible to detect landmark versions by looking at a time line of the updates to a file. We have seen that for many files, these updates are grouped into short barrages of edits separated by longer periods of stability. A good heuristic is to treat the newest version of each group as a landmark. Of course this heuristic may sometimes be wrong, so it may also be important to allow the user to specify other versions as landmarks.

Not all files exhibit a history of landmarks. Object files, for example, are of little interest after they have been linked to produce the final binary. We have observed that their histories are boring and the files are quite large. Object files and source files should thus be treated differently.

3.2. Elephant Cleaner Policies

In this section we discuss three cleaner policies we have examined for Elephant and implemented in our prototype. These three policies are listed below.

- Keep One
- Keep All
- Keep Landmarks

Keep One and *Keep All* are the simplest policies and represent the two ends of the retention spectrum. Keep One is equivalent to the standard file system model. There are many classes of files that this policy suits well. Files that are unimportant (e.g., files in /tmp, core files, etc.) or files that are easily recreated (e.g., object files, files in a Web browser cache, etc.) are good candidates for Keep-One retention. Similarly, Keep All is appropriate for files for which a complete history is important.

Keep Landmarks is a more interesting policy. The basic idea is to designate certain versions as landmarks and allow other versions to be freed as necessary. The key issue, however, is how to determine which versions are landmarks. One approach would be to require the user to designate the landmark versions. This approach, however, would interfere with our goal of freeing the user from direct involvement in storage retention. Instead, we permit the user to designate landmark versions and use a heuristic to conservatively tag other versions as possible landmarks. The cleaner then frees only versions that the policy determines are unlikely to be landmarks.

This landmark designation heuristic is based on the assumption that as versions of files get older without being accessed the ability of the user to distinguish between two neighbouring versions decreases. For example, we might designate every version of a file generated in the past week as a landmark. For versions that are a month old, however, we might assume that versions generated within one minute of each other are now indistinguishable to the user. If so, we can designate only the newest version of any such collection of versions to be a landmark, possibly freeing some versions for deletion.

Freeing a version in this way creates an ambiguity in the file's history as we have introduced a gap. A user may yet request the freed version of the file. The presence of this ambiguity is important information to the user. We thus retain information about freed versions of a file in its history and allow the user to examine the history for ambiguous periods of time. This information is important, for example, for the user to roll back a set of files to a consistent point in the past. The user can only be certain that the specified point is consistent if all files have an unambiguous version at that point in time.

To assist the user in locating consistent versions of a file group, we provide a utility for combining the version histories of a group of files to identify unambiguous periods in their combined history. If, for example, a user tries to rollback to "Jan 15 at 11:30:00" and a file version that existed at that time has been freed, this rollback is ambiguous. The user can use the version history tool to locate an unambiguous time (e.g., "Jan 15 at 11:25:00") and suitably refine her rollback. Of course, if the user had known that "11:30:00" was going to be important, she could have manually specified it as a landmark and thus prevented the system from creating the ambiguity.

Another solution to this problem is to clean files in groups specified by the user. We are investigating an extension to the Keep Landmarks policy that allows users to group files for consideration by the cleaner. The cleaner then ensures that every landmark version in the group is unambiguous with respect to all files in the group.

4. Implementation

4.1. Overview

We have implemented a prototype of Elephant in FreeBSD 2.2.7, which is a freely available version of BSD for personal computers. Elephant is fully integrated into the FreeBSD kernel and uses BSD's VFS/vnode interface. The standard UNIX file interface has been augmented with a new API to access the advanced features of Elephant, but all UNIX utilities work on current as well as older versions of files without requiring any changes.

4.2. Design

We define a version of a file to be its state after a *close* has been issued on it. Elephant's disk blocks are protected by copy-on-write techniques. Only the current version of a file may be modified. The first write to a file following an *open* causes its inode to be duplicated, creating a new version of the file. The first time an existing block is written after an open, the modified block is given a new physical disk address and the new inode is updated accordingly. All subsequent writes to that block before the close are performed in place. When the file is closed, the new inode is appended to an inode log maintained for that file. This inode log constitutes the file's history and is indexed by the time each inode in the log was closed. Concurrent sharing of a file is supported by performing copy-on-write on the *first* open and updating the inode log on the *last* close.

4.3. Elephant Meta-Data

Traditional file systems have one inode per name (but the reverse does not hold). Files can thus be uniquely and unambiguously named by their *inode number*, an index to their inode's disk address. Elephant departs from this model, because files have multiple inodes, one for each version of the file. To maintain the important naming properties of inode numbers, Elephant redefines inode numbers to index a file's inode log instead of its inode. Additionally, a level of indirection is introduced to provide flexibility in the location and size of inode logs. An inode number thus indexes an entry in the *inode map*, which stores the disk address of the corresponding inode log, as depicted in Figure 1.

The inode map contains the data needed to manage the file and its inodes. The two most important fields are the physical disk address of the inode log, and the file's *temperature*. The inode log is a temporally ordered list of inodes while the temperature is a heuristic used to prioritize the actions of the cleaner, as described below. The inode map is cached in memory and thus a file's current inode, which is always at the head of the inode log, can be retrieved with a single disk access.

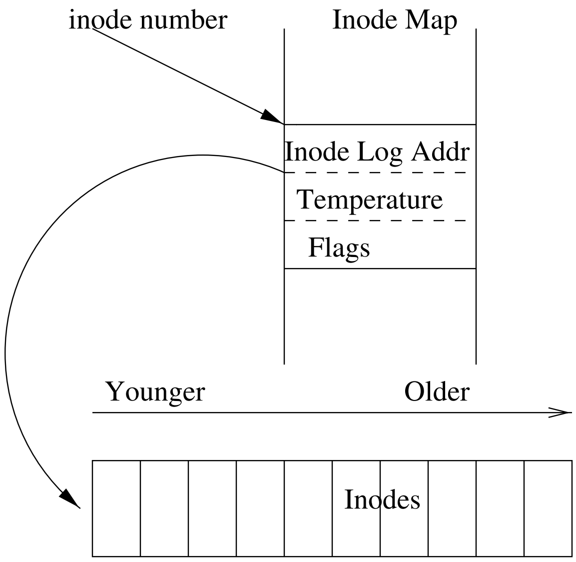

Figure 1. Inode Log

4.4. Directories

Directories map names to inode numbers. They differ in their usage from files in that files are explicitly opened and closed whereas directory modifications are implicit side effects of other system calls. For this reason Elephant handles the versioning of directories differently from that of ordinary files.

Elephant directories store versioning information explicitly and all versions of a directory are represented by a single inode. Each directory stores a name's creation time and, if deleted, its deletion time. It is possible for multiple instances of the same name to co-exist in a directory, provided that no two of them existed at the same time. Directory entires are retained as long as at least one version of the file they name remains in the file system.

An alternative to keeping versioning information in directories is to treat directories in the same fashion as files. The result, however, is wasteful of inodes and data blocks, because each name creation or deletion would require a new data block and thus a new inode for the directory.

4.5. API

Elephant allows users to add an optional timestamp tag to any pathname they present to the file system (e.g., when opening a file or when changing the current working directory). If this tag is present, Elephant accesses the version of the file that existed at the specified time. For example, if a user types "cd .#today:11:30", her working directory is changed to the version that existed at 11:30 of the current day.

When a timestamp tag is not specified, the selected version is determined by either the timestamp of the current working directory, if a relative pathname is specified (e.g., "file"), or the process's *current epoch*, if a complete pathname is specified (e.g., "/file"). Users can change the timestamp of their process's *current epoch* using the newly added *setepoch* system call. Child processes inherit their current epoch from their parent process when they are created.

The *readinodelog* system call was added to allow user-mode applications to access a file's history. This call returns a digest of the inodes in a file's inode log, including key fields such as the timestamp of each inode. As mentioned in Section 3, information about versions that Elephant's cleaner has purged is also returned by this call. We envision that this system call will be used by a new suite of utility programs that allow users to view and navigate file version histories; we have written some of these utilities, but this is an area of active research.

4.6. The Cleaner

The Elephant cleaner is responsible for reclaiming storage and is directed by the policies outlined in Section 3.2. It proceeds by picking a file to clean, reading its inode log, and selecting zero or more inodes to be reclaimed, compressed, or moved to tertiary storage (compression and tertiary storage are not supported by the current prototype, but are planned). To guide the cleaner to files likely to have the most reclaimable storage, Elephant maintains a *temperature* heuristic for each file; temperatures are stored in the memory-cached inode map. A file's temperature is recomputed when it is closed and when the cleaner examines the file. In the current prototype, temperatures are assigned using a heuristic based on the size of the file and the number of inodes in its inode log. We are actively investigating policy-specific heuristics that also consider the file's history profile, the time since the file was last cleaned, and other information gathered by the cleaner when it examines the file.

It is important to distinguish the Elephant cleaner from and the cleaner of a Log Structured File System. An LFS cleaner servers two roles: it frees obsolete blocks and it coalesces free space. In contrast, the Elephant cleaner's role is simply to free obsolete blocks. As a result, the Elephant cleaner has significantly lower overhead than an LFS cleaner, because Elephant's cleaning is performed without reading any file data blocks, only the inode log need be accessed. In contrast, the LFS cleaner must read every data block at least once, even obsolete blocks, and it may read and write active blocks multiple times.

5. Conclusions

We believe current file system models are flawed. Forcing the user to manage disk block reclamation and write-in-place file policies result in many problems. Previous systems that have attempted to solve these problems with snapshots or the retention of a limited number of versions fall short of what is required by users today.

Large cheap disk stores provide an opportunity for us to address the limitations of previous file systems. Users are quickly confused as multiple copies of their data accumulate, or become frustrated when they lose data in files from misbehaved applications or accidental deletion or overwrite. Elephant addresses these issues by providing a system where data blocks are immutable and the system decides when to deallocate disk blocks. The user need only manage the name space to keep their environment organized. Specifying backup policies at the granularity of files instead of the file system allows the file system to tailor its treatment of files depending on their type.

5.1. Status and Future Work

Our work on Elephant is proceeding along five fronts. First, we are extending our prototype to add support for compression and tertiary storage to the cleaner. Second, we are building a set of new utilities that exploit Elephant's novel functionality (we have already written a few utilities including "tgrep", "tls", and a history browser). Third, we are examining an alternate implementation that provides versioning at the level of blocks and abstracted by a logical disk [1]. Fourth, we are investigating how to backup an Elephant file system so that version histories can be recovered following a media failure. Finally, we are planning an extensive user study. To facilitate this study, we are modifying our prototype to allow it to shadow an NFS server. Users will thus be able to use Elephant without the risking their data to a research file system. This study will allow us to evaluate various cleaner-policy issues and to understand how user behavior changes when they use a file system that never forgets.

Acknowledgments

We would like to thank Jacob Ofir and Sreelatha Reddy who wrote some of the Elephant utilities and helped with experiements. Thanks also to Joon Suan Ong and Yvonne Coady who commented on earlier versions of this paper.

References

[1] W. de Jonge, M. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th Symposium on Operating Systems Princi ples*, pages 15–28, December 1993.

[2] Digital. *Vax/VMS System Software Handbook*. Bedford, 1985.

[3] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the Usenix Summer Conference*, pages 197–208, Boston, MA, June 1994. Usenix.

[4] D. Hitz, J. Lau, and M. Malcolm. File system design for a file server appliance. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 235–245, San Francisco, CA, January 1994. Usenix.

[5] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[6] L. Moses. *TOPS-20 User's manual*. USC/Information Sciences Institute, Internal manual, Marina del Rey, California.

[7] D. Presotto. Plan 9. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 31–38, Seattle, WA, USA, Apr. 1992. USENIX Association.

[8] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, Dec. 1975.

[9] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[10] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 25–34, Orcas Island WA (USA), Dec. 1985. ACM.

[11] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In USENIX Association, editor, *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 307–326, Berkeley, CA, USA, Winter 1993. USENIX.

[12] W. F. Tichy. RCS: A system for version control. *Software — Practice and Experience*, 15(7):637–654, July 1985.