17th ACM Symposium on Operating Systems Principles (SOSP '99),
Published as *Operating Systems Review*, **34**(5):110–123, Dec. 1999

Deciding when to forget in the Elephant file system

Douglas S. Santry[1], Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch[†],
Ross W. Carton, and Jacob Ofir

Department of Computer Science
University of British Columbia

{*dsantry,feeley,norm,carton,jofir*}*@cs.ubc.ca*

[†]Storage Systems Program
Hewlett-Packard Laboratories

aveitch@hpl.hp.com

Abstract

Modern file systems associate the deletion of a file with the immediate release of storage, and file writes with the irrevocable change of file contents. We argue that this behavior is a relic of the past, when disk storage was a scarce resource. Today, large cheap disks make it possible for the file system to protect valuable data from accidental delete or overwrite.

This paper describes the design, implementation, and performance of the Elephant file system, which automatically retains all important versions of user files. Users name previous file versions by combining a traditional pathname with a time when the desired version of a file or directory existed. Storage in Elephant is managed by the system using file-grain 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.

Elephant 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 file systems use disk stores. In particular, the current model of user-controlled storage management may no longer be ideal.

In a traditional file system, users control what is stored on disk by explicitly creating, writing, and deleting files. The key weakness of this model is that user actions have an immediate and irrevocable effect on disk storage. If a user mistakenly deletes or overwrites a valuable file, the data it stores is immediately lost and, unless a backup copy of the file exists, lost forever.

Over the years, a main focus of file system research has been to protect data from failure. Excellent solutions exist to protect data from a wide variety of network, system, and media failures. Users of an appropriately configured file system can now rest assured that their valuable data is safe and available, protected from all forms of failure; all, that is, but failures of their own making. As soon as users modify or delete a file, none of the carefully engineered protections in the file system can save them from themselves or from the applications they run. Some partial solutions and coping mechanisms do exist, but none adequately solves the problem.

Some early file systems such as Cedar provided a degree of protection from accidental overwrite, but not delete, by automatically retaining the last few versions of a file in copy-on-write fashion [20, 1, 13]. Limited storage space, however, meant that only a few versions could be retained. The choice of which version to prune was either left to the user or the oldest versions were deleted. In either case, valuable file versions could easily be lost.

Personal computer operating systems provide a degree of protection from accidental delete, but not overwrite, using the "trash can" metaphor, which requires a two step process to really delete a file. The trash can, however, provides only limited undo capability. Eventually storage becomes constrained and the trash can must be emptied. File deletions that occur shortly before the empty are afforded only a very limited period in which they can be undone.

In most well-maintained file systems, off-line backup storage is used to protect users from system failures, media failures, and their own mistakes. Checkpointing file systems such as Plan-9, AFS, and WAFL build this support into the file system using copy-on-write techniques to create periodic file-system checkpoints [16, 5, 6]. These checkpoints are available online and they allow users to access out-dated versions that were captured by a checkpoint. Changes that occur between checkpoints (or backups), however, are not recoverable. Furthermore, the fact that checkpointing occurs at the granularity of the entire file system limits the number and frequency of checkpoints.

The lack of file system support has required users and application developers to devise various coping mechanisms

[1]Author's current address is Network Appliance, Inc., Sunnyvale, CA.

to protect their data. A user who wants to maintain older versions of files must explicitly make and maintain multiple copies. A savvy user tends to be conservative, making many copies of data and avoiding deletes whenever possible. File-editing applications often provide operation-grain undo capability and perform live editing in a copy of the original file that replaces the file in an atomic operation when users commit changes.

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.

Furthermore, we believe that the amount of storage dedicated to files that are modified by users is growing slowly compared to the total amount of data stored in the file system. As the capacity of a single inexpensive disk approaches 50 GB, only a small fraction of this space will be occupied by files that require protection from user mistakes. The rest will be temporary, derived, and cached data that can be ignored, re-created, or re-fetched if lost.

This paper describes the design, implementation, and performance of the Elephant file system. In Elephant, old versions of files are automatically retained and storage is managed by the file system. Users specify retention policies for individual files, groups of files, or directories. The goal of Elephant is to allow users to retain important old versions of all of their files. User actions such as delete and file write are thus easily revocable by rolling back a file system, a directory, or an individual file to an earlier point in time.

2 Issues for storage management

A file system that protects users from their mistakes must separate storage management from the common file-system operations available to users (e.g., open, write, close, and delete). Deleting a file must not release its storage and file updates must not overwrite existing file data. To achieve this goal, the file system must retain sufficient information to be able to reconstruct old versions of files and directories. To undo an update, the previous content must be retained. To undo a delete, both the file's name and content must be retained.

It is obviously not feasible, however, to retain every version of every file. The system or user must decide what historic data can be discarded and when it should be freed. The key question is what are the respective roles of the system and the user in making this decision?

There are two competing factors to consider. First, increasing direct user control of reclamation tends to decrease the protection the user has from their mistakes. Storage reclamation is, by definition, an irreversible process. On the other hand, if the system controls reclamation, how do we ensure that the decisions it makes respect user needs? Only the user knows which versions of file data are important.

Solving this dilemma requires a compromise between the competing needs of user protection and user control. To understand how best to strike this compromise it is useful to review in more detail the different types of data users store in a file system, the different ways they access this data, and how these factors impact the question of what sort of history information the file system should retain. The remainder of this section summarizes our observations taken from several UNIX file system traces [14, 3, 18].

2.1 Not all files are created equal

The first key observation is that a file system stores many different types of files that each require different levels of versioning. The following simplified taxonomy demonstrates these differences.

- **Read-only** files such as application executables, libraries, header files, etc. have no version history.
- **Derived** files such as object files require no history or undo capability as they can be regenerated from their original sources.
- **Cached** files such as those maintained by a web browser require no versioning or history.
- **Temporary** files have short lifetimes and, while they may benefit from short-term histories for undo purposes if they are modified by users, long-term histories are not necessary.
- **User-modified** files may require histories, but even these files need varying degrees of versioning.

Clearly, a variety of file retention policies are required. User control can be achieved by allowing users to associate a policy with each of their files. User protection can be assured by placing the mechanism that implements these policies inside of the file system. In this way, a user can indicate that versioning is not required for some files and, for other files, the user can indicate what type of versioning is required. The next key question is thus what different types of versioning are possible and useful?

2.2 Goals: undo and long-term history

We break down the goal of protecting users from their mistakes into two related issues: (1) providing users the ability to *undo* a recent change and (2) maintaining a *long-term history* of important versions. The two differ in the amount of time they cover and in the completeness of the history they retain.

Undo requires that a complete history be retained, but only for a limited period of time. The history must be complete, because there is typically no good way to predict which actions a user might later want to undo. Limiting the amount of storage needed to support undo thus requires restricting the period of time in which an undo is possible. The system might, for example, allow the user an hour, a day, or a week to realize that an operation was in error. Within that interval, the user can reverse any delete or overwrite. Once that second-chance interval passes, however, changes become permanent.

Long-term history serves a different purpose. Once the undo period has passed, it may still be necessary to retain certain important versions of a file. It is typically not appropriate or useful to retain every version of the file. Instead, users select key landmark versions to be retained long term. In a software revision control system [17, 24], for example, users select landmark versions by checking them into a repository. Intermediate changes the user makes in their working directory are not retained.

These two goals are closely related. It may be useful to have intermediate regimes in which some undo's become impossible, but more versions are retained than just the landmark versions. Additionally, the importance of a landmark version may degrade over time to the point where some old landmarks can be deleted.

2.3 Pruning the long-term file history

The final remaining issue is how to prune the long-term history and how to identify landmark versions. To understand this issue it is helpful to review how users use current file systems and how increasing disk storage capacity impacts the decisions they make.

As stated previously, users cope with the fragility of current file systems by making frequent copies of important data. Data that is changed frequently is copied frequently. Furthermore, users are reluctant to delete files, because doing so permanently removes them from the file system.

Ironically, as storage becomes larger, it becomes more difficult for users to manage the versions they create. When storage is fairly constrained, users are required to frequently assess the files that they are maintaining, and to 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.

When it becomes 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. Unfortunately, the user may have no good way to tell which old version is important.

Any solution that relies solely on the user to identify landmark versions is problematic, because failure to identify an important version as a landmark can result in the loss of important data. It is thus important for the file system to assist the user in detecting landmark versions.

Fortunately, it is often possible to detect landmark versions by looking at a time line of the updates to a file. In our study of UNIX file-system traces, 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, in addition to any landmarks explicitly identified by the user. Another option is to treat as a landmark any version with a large delta to the subsequent version, i.e., where the amount of work involved to recover the older version would be large.

2.4 Summary

In summary, the following issues are key to the implementation of a file-system storage management mechanism that protects users from their mistakes.

- Storage reclamation must be separated from file write and delete.
- Files require a variety of retention policies.
- Policies must be specified by the user, but implemented by the system.
- Undo requires complete history for a limited period of time.
- Long-term histories should not retain all versions.
- The file system must assist the user in deciding what versions to retain in the long-term history.

3 The Elephant file system

This section describes the design of the Elephant File System and the file retention policies it uses to manage storage.

3.1 Design overview

The key design principle of Elephant is to separate storage management from the common file system operations available to users (e.g., open, write, close, and delete). The file system, not users, handles storage reclamation. 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 when it is written.

File versions are named by combining the file's pathname with a date and time. Versions are indexed by their creation time (i.e., the file's modification time) and versioning is extended to directories as well as files; only the current version of a file can be modified. The system interprets the pathname-time pair to locate the version that existed at the specified time. A file version thus does not have a unique name; any time in the interval in which a version was current can be used to name the version.

By rolling a directory back in time, a user can see the directory exactly as it existed at that earlier time, including any files that the user subsequently deleted. Delete thus becomes a reversible name-space management operation.

Storage is managed by the system using retention policies associated with each file. Users can specify policies for a file, a group of files, or for a directory; newly created files by default inherit the policy of the directory in which they are created. Periodically, a file system cleaner process examines the file system and uses these policies to decide when and which disk blocks to reclaim, compress, or move to tertiary storage. This approach allows users to indicate which files require versioning and what sort of versioning they require. As a result, the majority of files can be handled by keeping a single version with no history, as is done today, while providing detailed history information for only the files that require it.

3.2 File retention policies

We have identified the following four file retention policies for Elephant and have implemented these policies in our prototype.

- Keep One
- Keep All
- Keep Safe
- Keep Landmarks

Keep One provides the non-versioned semantics of a standard file system. Keep All retains every version of the file. Keep Safe provides versioning for undo but does not retain any long-term history. Keep Landmarks enhances Keep Safe to also retain a long-term history of landmark versions. The remainder of this section discusses the details of these policies.

3.3 Keep One: no versioning

Keep One provides semantics identical to a standard file system, retaining only the current version of a file. Deleting a Keep-One file removes the file immediately. Writing a Keep-One file updates the file in place; copy-on-write is not used. An important property of Keep One is that it is the only policy that allows users to directly control storage reclamation.

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.

We anticipate, however, that most files that users modify directly, even temporary files, will have at least Keep Safe semantics.

3.4 Keep All: complete versioning

Keep All retains every version of every file. On each open a new version of the file is created, and on each close that version is finalized. While we do not believe that there are many files that require every version to be retained, Keep All is available if necessary.

In addition, Keep All is the basis of the Keep Safe and Keep Landmarks policies, as file versions are created in exactly the same way as in Keep All; these policies differ only in when versions are forgotten.

3.5 Keep Safe: undo protection

Keep Safe protects users from their mistakes but retains no long-term history. The policy is parameterized by the length of the *second-chance interval*. The system guarantees that all file-system operations can be undone for a period defined by this parameter. A second-chance interval of seven days, for example, means that the user can rollback any change that occurred within the past seven days. Any change that occurred more than seven days ago, however, becomes permanent.

Implementing Keep Safe requires that every version of a file be retained until it is no longer needed for rollback. Notice that simply retaining a version until it is older than the second-chance interval is not sufficient. Instead, a version must be retained until a *younger* version, or the file's deletion, becomes permanent. For example, if a file with a year-old version is modified or deleted, the year-old version must be retained until the second-chance interval for that modification or deletion expires.

3.6 Keep Landmarks: long-term history

Files that store important user information often require histories. For these files, in addition to the protections of Keep Safe, the system also maintains a long-term history.

Keep Landmarks retains only landmark versions in a file's history; non-landmark versions are freed as necessary to reclaim storage. The key issue, however, is how to determine which versions are landmarks. We follow a combined approach that provides users with direct control, but also seeks to protect users from their mistakes. A user can specify any version as a landmark. In addition, the system uses a heuristic to conservatively tag other versions as possible landmarks. The cleaner frees only versions that the Keep-Landmarks 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 neighboring 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 non-landmark versions introduces discontinuities in a file's history. A user may request the freed version of the file. The presence of these discontinuities 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 discontinuities. 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 the histories of all files are continuous at that point in time.

Finally, the Keep-Landmarks policy allows the user to group files for consideration by the policy. Grouping is important for inter-dependent files whose consistency requires viewing all files as of the same point in time. When this type of inter-dependency exists, the files are inconsistent if a discontinuity exists in any of the files, because the missing version is required for a consistent view. Keep-Landmarks solves this problem by ensuring that no file in a group has a discontinuity that overlaps a landmark version of any other file in the group. This grouping can be done at a number of granularities ranging from explicitly indicating which files are interdependent to specifying directories or entire subtrees that should be treated as a group.

4 Prototype implementation

We have implemented a complete prototype of Elephant in FreeBSD 2.2.8, 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 is augmented with a new API to access the advanced features of Elephant, but all UNIX utilities work unmodified on current as well as older versions of files.

This section describes the design and implementation of our prototype, detailing the file system's meta data, versioned directory structure, enhanced kernel interface, storage reclamation mechanism, and user-mode tools.

4.1 Overview of versioning

Elephant versions all files except those assigned the Keep-One policy. Disk blocks of versioned files are protected by copy-on-write and only the current version of a file can be modified. Keep-One files can be updated in place, following the same procedure as a standard UNIX file system.

A version is defined by the updates bracketed between an open and a close. The first write to a versioned 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. This copy-on-write is avoided for writes that only append to the file. All subsequent writes to a modified block before the *close* are performed in place. When the file is closed, the 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 last modified. 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.

To economize disk space, files that have no history are stored in a standard inode instead of an inode log. The system adaptively switches between an inode and inode log as necessary. For example, a newly created Keep-Safe file is stored in a single inode. When the file is modified, an inode log is created for the file to store both the original and new versions of the file. When the current version is sufficiently old, the cleaner frees all historic versions and the system frees the file's inode log and replaces it with a single inode.

Note that the inode log maintains version history based on a file's inode and not its name. This level of versioning does not easily capture namespace changes such as those that occur when a file is created, deleted or renamed. Renaming a file, for example, changes its name but not its inode. To simplify namespace journaling, Elephant stores directories as name logs. Name logs are used in conjunction with inode logs to provide users with an alternative view of a file's history based on file name. Name histories lists all versions associated with a name, even those that belong to different inodes (or inode logs).

Name histories are useful, for example, for programs such as Emacs that overwrite files by writing to a new inode that is then renamed to the original name. Using this approach, every new version of a file is stored in a different inode (or inode log) and thus the file's history is available only in the name-centric view. While we believe that this style of access is likely to disappear once versioning file systems such as Elephant become widespread, the current prototype does allow users to view both inode and name histories for any file. The file retention policies implemented in the prototype, however, use only inode histories.

4.2 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 *inumber*, an index to their inode's disk address. Elephant departs from this model, because files can have multiple inodes, one for each version of the file. To maintain the important naming properties of inumbers, Elephant redefines inumbers to index an *imap*.

4.2.1 The imap

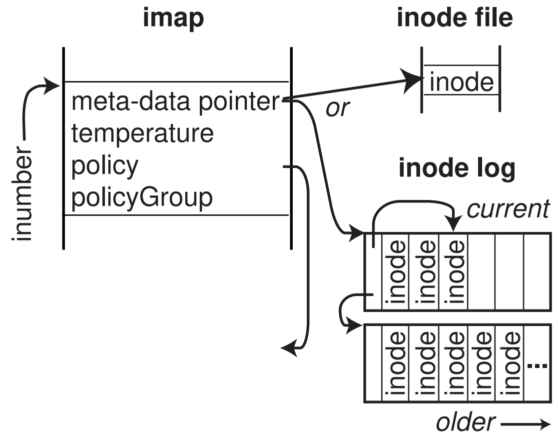

Figure 1. Meta data: the imap, inode logs, and the inode file.

As depicted in Figure 1, the imap provides a level of indirection between an inumber and the disk address of a file's inode or inode log. There is an entry in the imap for every file. Each entry is 16 bytes, storing the file's inode or inode log pointer, *temperature*, *policy*, and *policyGroup*. The imap is stored on disk and cached in memory; its disk address is recorded at a fixed location on disk.

The meta-data pointer has two parts: *type* and *address*. *Type* indicates whether the file is stored in an inode or an inode log. For inodes, *address* is the offset of the inode in the *inode file* and, for inode logs, *address* is the disk block number of the file's inode log. The inode file and inode logs are described in detail below.

Temperature is a heuristic used to guide the system cleaner to files likely to have the most reclaimable storage. The temperature encodes a value and an expiration date; the system artificially raises the value of an expired temperature to ensure that the cleaner re-examines the file. A file's temperature is increased by the system when the a new version is created by an amount dependent on the number of blocks that were copied-on-write. It is reset by the file's policy module when the cleaner examines the file, since at that instant there should be no reclaimable storage in the file. Policy modules that base their reclamation decisions on elapsed time use the timeout value to expire obsolete temperature estimations. For example, the Keep Safe policy will want to re-examine a file when its second-oldest version is as old as the retention period is long.

Finally, *policy* names the file's storage retention policy and *policyGroup* organizes files that are grouped by this policy as described in Section 3.6. The imap entries for a group's files are linked into a circular list; policyGroup stores the inumber of the next file in the group.

4.2.2 Non-versioned inodes

Elephant stores inodes for non-versioned files in a distinguished file called the *inode file*. This approach differs from traditional UNIX file systems, which pre-allocate inodes at fixed disk locations. The inode file itself is stored in an inode log, not an inode, in order to avoid circularity. Our approach has the advantage that inode storage space need not be pre-allocated when the file system is initialized. The inode file can grow and shrink as easily as any other file. A new inode is allocated by picking a random entry in the inode file and scanning forward until a free inode is found. If a free slot is not found after scanning a bounded number of inodes, a new block is added to the inode file and the inode is allocated there. The inode file can be compacted by moving inodes to create empty blocks and then removing those blocks from the file.

4.2.3 Inode logs

Inode logs are allocated and named at the granularity of disk blocks and they thus consume considerably more storage than a single inode, which consumes only 168 bytes. An inode log stores the versions of a file as an ordered list of inodes. If the log spans multiple blocks, the blocks are linked together in descending-chronological order, with the block containing the current inode at the head of the list. The file's imap entry stores the address if this block and the block stores the offset to the current inode. Reading the inode of the current version of a file is thus as efficient as reading the inode of a non-versioned file in Elephant or any standard UNIX file system. Older versions are located by a linear scan of the log.

The inode log also uses inodes to record information about reclaimed versions and to record a file's deletion time. Recall that we retain information about reclaimed versions so that users can detect discontinuities in a file's history. To do this, the system replaces reclaimed inodes with an inode that describes the discontinuous interval they leave behind. Similarly, when a file is deleted, an inode is added to the end of the log to record the delete time. The delete time is needed by the Keep-Safe policy, for example, to determine when the delete becomes permanent.

4.3 Directories

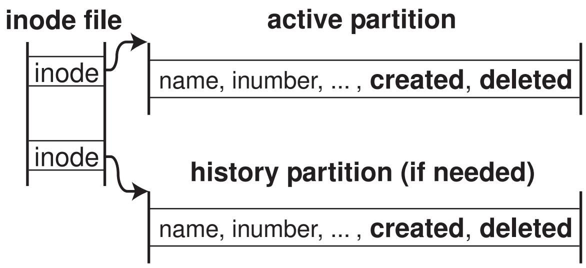

Figure 2. Directory structure.

Directories map names to inumbers. 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 so are implemented using standard inodes, not inode logs. Each directory entry 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. A name remains in a directory as long as at least one version of the file it names remains in the file system. Directory layout is depicted in Figure 2.

Initially a directory is stored in a single inode. If a directory accumulates a sufficiently large number of deleted names, a second inode is allocated to store the deleted names. As necessary, the system periodically moves deleted names from the active inode to the history inode and compacts the active inode. A name lookup on the current version of a directory scans only the active node, while a lookup on an past version of the directory scans both inodes. This partitioning thus ensures that the history information stored in a directory does not significantly slow the common case operation of looking up a name in the current version of the directory.

An alternative to keeping versioning information in directories would have been to treat directories in the same fashion as versioned files. The result, however, would be 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. Copying a large directory, for example, would create a new directory inode for every name in the directory.

4.4 System interface

Elephant allows users to add a timestamp tag to any pathname they present to the file system. If this tag

setCurrentEpoch	(timestamp)
getHistory	(file) $\Rightarrow$ history
setLandmark	(file)
unsetLandmark	(file)
setPolicy	(file, policyID)
groupFiles	(fileA, fileB)
ungroupFile	(file)

Table 1. Kernel interface for user-mode applications.

mapImap	(pathname) $\Rightarrow$ mntPt
lockFile	(mntPt, iNumber)
unlockFile	(mntPt, iNumber)
readBlock	(mntPt, block)
writeBlock	(mntPt, block)
freeBlock	(mntPt, block)

Table 2. Kernel interface for the cleaner process.

is present, Elephant accesses the version of the file that existed at the specified time. If a user types "`cd .@12-nov-1999:11:30`", for example, their working directory is changed to the version that existed at the specified date and time.

If 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 a process's current epoch as described next and child processes inherit their parent's current epoch when they are created.

Table 1 shows the seven new system calls added for Elephant. The **setCurrentEpoch** operation changes the current epoch of the calling process. The **getHistory** operation returns a digest of the inodes in a file's inode log, including key fields such as the timestamp of each inode. The **setLandmark** operation designates the specified file version to be a landmark so that the Keep-Landmarks policy will never remove the version; **unsetLandmark** removes this designation. The **setPolicy** operation assigns a retention policy to a file. The **groupFiles** operation places **fileA** and **fileB** in the same group for treatment by their retention policy, as described in Section 3.6. The new group contains the transitive closure of any files previously grouped with either of the two files. The current prototype requires that the user explicitly create file groups by naming each file. We are considering more flexible group creation schemes where directories or subtrees could be grouped together. The **ungroupFile** operation removes **file** from its group, if one exists.

4.5 Storage reclamation

Storage reclamation is handled by a system cleaner that implements the per-file retention policies described in Section 3.2. The cleaner is a privileged and trusted user-mode process that uses a customized kernel interface to interact with the Elephant file system.

Table 2 details the kernel interface used by the cleaner. The **mapImap** call maps the inode map for the Elephant file system named by **pathname** into the cleaner's virtual address space. Also, it returns **mntPt**, the file system's mount-point id, which the cleaner uses on subsequent calls to the kernel. The mount-point id is necessary to allow multiple Elephant file systems to co-exist on one system.

The **lockFile** and **unlockFile** operations lock and unlock the selected file's inode log; no other process is permitted to operate on the file while the cleaner has it locked. The cleaner reads and synchronously writes the inode log directly using the **readBlock** and **writeBlock** calls; blocks are named by mount point and relative disk address. Finally, **freeBlock** frees the specified physical block.

The cleaner process runs in the background, scanning the memory-cached imap for a high-temperature file. When it selects a file, it calls **lockFile** and then uses **readBlock** to read the file's inode log into memory. It then scans the log to build a binary tree of the physical blocks allocated to the file and counts the number of inodes that reference each block. If the **policyGroup** field in the imap indicates that the file is part of a policy group, it repeats this process for every file in the group. To avoid deadlock, if the cleaner encounters a file that is already locked, it immediately releases all of its locks and skips the group.

The cleaner proceeds by invoking the file's policy to pick zero or more inodes for deletion. It deletes each inode by updating the in-memory inode log and decrementing the block-list reference counts for each of the inode's blocks. Finally, it calls **writeBlock** to write the modified inode log to disk, calls **freeBlock** to free any block in the block list with a reference count of zero, and calls **unlockFile** to release its lock on the file's inode log.

The cleaner protects the file system from failures that occur during cleaning by ordering disk writes to ensure that a block is not freed if any inode on disk stores a reference to it. A recovery procedure reclaims disk blocks that are unreachable from any inode. If the cleaner process fails independently of the rest of the system, locked inodes are inaccessible and partially freed disk blocks are not reclaimed. The system detects and recovers from cleaner failure by terminating the cleaner process, releasing all inode-log locks it holds, scheduling the recovery procedure, and restarting the cleaner.

Finally, it is important to distinguish the Elephant cleaner from the cleaner of a Log Structured File System [19, 22, 9]. An LFS cleaner serves 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 typically read every data block at least once, even obsolete blocks, and it may read and write active blocks multiple times.

4.6 Tools

We have augmented the standard set of UNIX file manipulation tools to add several new tools that manage the file system's new temporal dimension. The **tls** tool is similar to **ls**,

registerPolicy	(pathname)	⇒ policyID
unregisterPolicy	(policyID)	
lookupPolicy	(policyID)	⇒ pathname

Table 3. Kernel interface for registering application-defined policies modules.

cleanFile	(fileHistory)	⇒ (verList, newTemp)

Table 4. Interface exported by application policy modules; called by the system cleaner.

but instead outputs a list of a file's history, including information about deleted versions. The tool allows users to select a history of a specified inumber or name; name histories may include versions from multiple files with different inumbers. Similarly, **tgrep** searches for patterns in every version of a file in much the same way that **grep** searches a list of files. Finally, **tview** provides a graphical view of the individual or combined histories of a group of files.

5 Application defined retention policies

Elephant allows applications to define and implement their own file retention policies to augment the system defined policies described in Section 3.2. An application defined policy can be used, for example, where the user wishes to use a different definition of landmark versions — perhaps one based on how much file data changed between versions rather than on the timeline of those changes. Application-defined policies run as non-privileged and untrusted processes that communicate with the privileged system cleaner.

5.1 Registering an application policy

Application-defined policy modules must be registered with the kernel before they can be used. When registered, the kernel assigns a unique number to the policy and users call the standard **setPolicy** operation to assign this policy to files. The system cleaner invokes application-defined policies in much the same way it invokes the system-defined policies.

Table 3 shows the kernel interface for registering application-defined policy modules. Applications call **registerPolicy** to register the program specified by **pathname** as a new policy and they call **unregisterPolicy** to remove a policy. Policies can be unregistered only by a system administrator or by the user that registered the policy. The kernel assigns a unique **policyID** to all policies and stores the registration persistently.

The system cleaner calls **lookupPolicy** to locate the pathname of the policy module for a given policyID. If the policy has been unregistered, **lookupPolicy** returns an error and the file's policy defaults to the Keep All system policy.

5.2 Operation of application policies

Application-defined policy modules export the single **cleanFile** operation shown in Table 4. When the system cleaner encounters a high-temperature file handled by an application-defined policy, it forks the appropriate policy-module program, unless it is already running, and calls the process's **cleanFile** operation. This process runs with the privileges of the target file's owner. To amortize the overhead of starting the policy-module program, the cleaner maintains a pool of recently accessed processes. It limits the total number of processes, terminating the least-recently accessed process when this limit is reached.

Before calling **cleanFile**, the system cleaner locks the target file's inode log, reads its inode log into memory, and builds its allocated-block list, as described in Section 4.5. In the call to **cleanFile**, the cleaner provides the policy module with a digest of the file's history that includes block-allocation information.

In response to the **cleanFile** call, the application policy module examines the file's history and builds a list of the zero or more versions of the file it wishes to delete. It computes a new temperature for the file and returns the **verList** and **newTemp** to the system cleaner. This procedure is the only way that an application-defined policy can change a file's temperature. Unlike system policies, application policies do not get control when a file is closed and thus they can not affect the file's temperature at that time.

Upon successful completion of a **cleanFile** call, the system cleaner deletes the versions listed in **verList**, frees the associated disk blocks, updates the file's temperature, and unlocks its inode log, as described in Section 4.5. If **cleanFile** fails to return within a prescribed timeout interval, the system cleaner cancels the call, and unlocks the file's inode log. This timeout protects the system from policy-module failure.

6 Performance and evaluation

This section assesses the feasibility of our design. First, we compare the performance of our Elephant prototype to the standard FreeBSD FFS file system. Second, we examine the types of files stored by a large file system to estimate what portion of its files would be versioned. Third, we analyze file-system trace data to estimate how much extra storage an Elephant file system might consume for history information. Finally, we describe a key feature of the Elephant prototype that allows it to shadow an NFS server and we report on the results of two short-term user studies conducted using this feature.

6.1 Prototype performance

Our experimental setup consists of a 200 MHz Pentium MMX with 64 MB of RAM and a 3 GB IDE disk. Our prototype is a modified FreeBSD 2.2.8 kernel. We compare the performance of the prototype to an unmodified FreeBSD 2.2.8 kernel, which uses the UNIX fast file system (FFS) [11]. Both file systems were configured with a 4 KB block size. The measurements were taken by configuring each file system in the same disk slice to normalize for the effects of disk geometry.

In the following tables, **FFS** refers to the fast file system, **EFS-O** refers to files using the Keep-One policy whose in-

Operation	EFS-V (μs)	EFS-O (μs)	FFS (μs)
open (current)	134	133	132
open (20th ver)	140	—	—
open (75th ver)	160	—	—
write (4 KB)	58.7	54.5	47.3
close (upd)	35.8	34.5	34.9
close (upd 24th ver)	121	—	—
create (0 KB)	5040	3750	3930
delete (0 KB)	452	2154	3010
delete (64 KB)	446	4522	4732

Table 5. Performance of basic file system operations.

odes are stored in the inode file, and **EFS-V** refers to files using the Keep-All policy whose inodes are stored in inode logs (the performance of Keep Safe and Keep Landmarks is identical to Keep All).

6.1.1 *Micro-benchmarks*

Table 5 summarizes the performance of the basic file system operations. All measurements were taken by running a user-mode program and using the Pentium cycle counter to measure the elapsed time each system call spends in the kernel. The times do not include the overhead of entering and exiting the kernel. Each number represents the median of 10,000 trials.

The first three lines of Table 5 show the time to open a file when its inode (or inode log) is cached in memory. The first line shows that the time to open a file's current version is virtually the same in Elephant and in FFS. This is also true if the inode and inode logs are not cached; in this case, both EFS and FFS are slowed by the latency of a single-block disk read. The next two lines show the cost of opening an older version, which requires scanning the inode log and possibly reading additional inode-log blocks; each 4 KB inode-log block stores 24 inodes. Opening the 20th version is 6 μs slower than opening the current version, 0.3 μs for each inode scanned. Opening the 75th version is 26 μs slower. If the inode and inode-log blocks are not cached in memory, **open (20th ver)** requires one disk read and **open (75th ver)** requires four, one for each inode-log block it must scan.

The cost of the first write of a 4 KB block after an open is 58.7 μs in EFS-V and 47.3 μs in FFS, including the time to schedule an asynchronous disk write. EFS-V is 11 μs slower due to the copy-on-write operation required for this first write. The performance of subsequent writes to the block before the close is the same as in EFS-O. EFS-O is 7.2 μs slower than FFS, because the prototype currently performs a modified copy-on-write for these writes as well. This use of copy-on-write is an artifact of the way our prototype evolved; it is not required for versioning and could be replaced with an update-in-place scheme with the same performance as FFS.

The next two lines of Table 5 show the time to close a modified file. For EFS-V, the close produces a new version of the file. If the file's new inode fits in the current inode-log block (i.e., **close (upd)**) the performance of EFS-V is virtually the same as EFS-O and FFS. If it does not fit (i.e., **close (upd 24th ver)**) a new inode-log block is allocated and overhead increases by 85.2 μs. All EFS and FFS close operations require a single disk write; in the times reported, this write is handled asynchronously. Finally, if the inode or inode log is not cached in memory, all EFS and FFS close operations require one disk read. Closing a read-only file is the same in EFS and FFS.

The last three lines of Table 5 show the time to create and delete a file. Creating an empty versioned file requires 5.04 ms. This time is 1.11 ms slower than FFS, because EFS must allocate a new inode log for the file and write it to disk. Deleting a versioned file in EFS is considerably faster than FFS, both because an EFS delete does not release the file or its disk blocks and because EFS writes some meta data asynchronously. The actual release of storage in EFS is performed by the cleaner process, and Section 6.1.4 reports on its performance.

Finally, we measured the performance impact of versioned directories. Recall that an Elephant directory stores both active and deleted names and that a second inode is used to archive deleted names. The system periodically moves deleted names to the archive and compacts the active names; the overhead of this operation is 0.22 μs per name in the active inode. Name lookup in the current version of a directory is slowed by the number of deleted names in the active inode that must be skipped. Currently, we trigger compaction when the active inode has 20% deleted names. The impact on name-lookup time, however, is much less than 20%, because a deleted entry can be skipped without performing a name comparison, the operation that dominates lookup overhead.

6.1.2 *The Andrew file system benchmark*

We ran the modified Andrew File System Benchmark for EFS-V, EFS-O, and FFS. This standard benchmark is designed to represent the actions of a typical UNIX user. It creates a directory hierarchy, copies 70 source files totaling 200 KB into the hierarchy, traverses the hierarchy to examine the status of every file, reads every byte of every file, and compiles and links the files.

To be conservative, we modified the prototype to force all EFS-V files to be stored in an inode log. We did this because Andrew does not overwrite or delete files and thus the EFS adaptive meta-data allocation scheme would normally store these single-version files in an inode, not an inode log.

The benchmark's elapsed time was 19 s for EFS-V and EFS-O, and 18 s for FFS. FFS was one second faster in the compile-and-link phase of the benchmark. The total meta data consumed by files was 18 KB for EFS-O and FFS and 444 KB for EFS-V. EFS-V consumed 426 KB more space, because it stored files in 4 KB inode logs instead of 168 B inodes.

6.1.3 *Copying a large directory*

We also measured the performance of copying a directory that is substantially larger than that used in the Andrew

Benchmark. For this experiment, we copied the FreeBSD-kernel source tree, which consists of 1525 files totaling 20 MB. Again, we forced all EFS-V files to use an inode log.

We measured an elapsed time of 49 s for EFS-V, 56 s for EFS-O, and 115 s for FFS. EFS is faster than FFS because it performs meta-data writes asynchronously and because EFS can place a newly created inode or inode log anywhere on disk and is thus able to group meta-data and data writes more effectively than FFS. The total meta-data consumed in EFS-O and FFS was 0.24 MB; EFS-V consumed 5.9 MB.

6.1.4 Cleaner performance

To measure the ability of the cleaner to free disk blocks, we created three versions of 1000 different files. Each version modified every block of a file. File sizes ranged between 1 KB and 1 MB, with most of the files less than 4 KB; the mean file size was 145 KB. The Keep-Safe policy was assigned to each file and the cleaner was triggered when the changes became permanent. The cleaner then removed two versions of each file for a total of 284 MB.

In this experiment, the cleaner was able to release storage at a rate of 5.6 MB/s. The cleaner spent roughly 70% of its time calling **freeBlock**, once for each block it frees. An optimization that allowed blocks to be freed in bulk would thus substantially improve cleaner throughput.

6.2 File system profile

We have examined the profiles of the files stored in the home-directory file system of a large workgroup server within HP Labs. The server currently supports approximately a dozen active researchers, who use it for development, document preparation, email, etc. This single file system contains approximately 15 GB of data in 360,000 files and 27,000 directories.

Using rough heuristics based on the file extension and the results of the UNIX *file* command, we divided these files into the following major categories.

- **Source** includes program source files, i.e., C, C++, perl, shell scripts, etc.
- **Document** includes files used for general document preparation, typically plain text, HTML, word processor files, and mail files.
- **Derived** includes files that are derived from other files, and which can presumably be easily re-created. These include object, library, executable, postscript, and PDF files.
- **Archive** includes those files that are typically used for archival purposes, such as tar and compressed files and data files containing experimental results etc.
- **Temporary** includes files with extensions like .tmp and Netscape cache files.
- **Other** includes all other files. Unfortunately, it was impossible to categorize all files effectively (there were over 4,000 distinct file name extensions present in the file system).

File Type	Files (%)	Bytes (%)
Source	14.6	3.4
Documents	22.6	11.0
Derived	20.6	53.3
Archive	3.9	28.5
Temporary	13.0	3.0
Other	25.2	0.8

Table 6. File distribution of 15 GB HPL server home directories.

Table 6 shows the distribution of files into these six categories. It is reasonable to assume that most Derived and Temporary files would use the Keep-One policy, most Archive files would use the Keep-Safe policy, and that the remainder might use the Keep-Landmarks policy. If these assumptions hold, then the distribution of files to policies breaks down as follows:

- **Keep One**: 33.6% of files – 56.3% of bytes
- **Keep Safe**: 3.9% of files – 28.5% of bytes
- **Keep Landmarks**: 62.4% of files – 15.2% of bytes

These results are conservative, as it is likely that many of the files we assigned to the Keep-Safe and Keep-Landmark categories would actually be in Keep One. A small sampling of these files reveals that many of them are part of packages from other sources (e.g. gcc distributions) that are essentially read-only.

A larger scale study of file-system contents on a variety of machines at Microsoft [2] reveals results similar to those above. Using the file name extension information they provide, and making an assignment of extension type to policy, we determine that approximately 12% of the bytes in their file systems would probably be under the Keep Landmarks policy.

Another interesting result of the Microsoft study is their observation that file systems are, on average, using only 50% of the available disk capacity. This implies that modern systems have sufficient free capacity to keep numerous versions of many files.

6.3 Trace studies

We also collected file-system trace data from the same server at HP labs that we used for our file-system profile. Using an HP-UX system call tracing facility, we recorded all file system activity that occurred between August 29 and October 8, 1999. In particular, we recorded all open, close, read and write events on all files in the system.

We have used these traces to try to provide an approximate upper bound on the growth of an Elephant file system. We processed the traces by generating a list of modified or updated files, and categorizing these into one of our three policy categories using the same heuristic as we used for the file-system profile described in Section 6.2.

Policy	Files (%)	Bytes (%)	Writes (% Bytes)
Keep One	33.6	56.3	98.7
Keep Safe	3.9	28.5	0.6
Keep Landmarks	62.4	15.2	0.7

Table 7. Estimated distribution of files and file-writes to Elephant policies.

Using trace data for the week of Monday, September 27 through Friday, October 1, we calculated the distribution of writes into each of these categories. The average amount of data written each day was 112,219 KB, of which the vast majority, 110,793 KB (98.7%), was to files in the Keep One category, 662 KB (0.6%) was to Keep Safe category files, and 764 KB (0.7%) was to Keep Landmark files. Table 7 summarizes the results of both the profile and trace studies.

These results are very promising, for three reasons. First, the amount of data needing some form of versioning per day (1.4 MB) is not large: it is only a small percentage of modern disk drive capacity. Second, this is a conservative figure. Several of the files included in the final traces would not be versioned in an Elephant system, as they were copies of other existing files (e.g. from source packages downloaded from the internet). Finally, the analysis above does not account for file deletion or overwrites. It is certain that these operations would reduce the amount of storage growth, as Elephant forgot some of the file versions.

6.3.1 Impact on the buffer cache

A potential drawback of Elephant is that it reduces the effectiveness of buffer-cache write absorption and thus increases the number of disk writes. In most UNIX systems, two writes to the same file system block within a certain time period (typically thirty seconds) will be absorbed by the buffer cache, and will result in only a single disk write. In addition, writes that occur shortly before a delete need not be written to disk. For Elephant versioned files, however, if there is an intervening close between the file system writes, then two separate disk writes are required. Similarly, writes to a deleted versioned file must be written to disk.

We examined our file system traces to determine the potential increase in write traffic. Only a very small proportion (less than 5%) of the overall writes are to blocks that are overwritten, or to files that are are deleted, within 30 seconds of that write. Factoring in our observation that only 1.3% of writes are to potentially versioned files, we conclude that their impact on disk write traffic should be minimal. A qualitative analysis of a subset of the traces showed that the few writes in this category were typically to files being actively edited, where the user had performed two or more "save" operations in a short space of time.

6.4 NFS shadowing and user studies

Convincing people to trust their valuable data to a research file system is a daunting task at best, and for a good reason. A key part of the evaluation of our ideas, however, requires that people use the system to do real work.

To solve this dilemma, we modified our prototype to shadow NFS traffic on our local network. The modified prototype snoops NFS traffic between clients and NFS servers and duplicates all file and directory creations and deletions and all file writes in an Elephant file system. To collect information about how users access old files, the prototype logs each time an old version of a file is accessed.

As a result, people can keep their files on an NFS server and perform all updates over NFS, while being able to view the history and change the retention policy of the shadowed copy of their files stored by Elephant. Furthermore, we can gain confidence in the robustness of the prototype by exercising it with a real, high-volume workload.

The NFS-shadowing implementation was complicated by the fact that NFS clients do not inform the NFS server when a file is closed. This information is critical, however, because closes determine when a new version is created. To solve this problem, we used a heuristic that assumes a file was closed if a pair of writes is separated by more than 10 seconds.

We have conducted two preliminary user studies using this technique, one with a group of eight graduate-students in a research lab, the other with a class of twenty students working on programming assignments in a graduate course. The first study covered roughly four weeks and the second covered about two weeks while students were working on one assignment. In both studies, we marked all files as Keep All and tracked which old versions were accessed by users and when they were accessed.

As a result of these two studies we have gained confidence in the stability of our prototype and we have seen that users will access old versions of their files if the file system retains them. We have also seen, however, that short-term studies can not provide useful information about the efficacy of long-term policies such as Keep Landmarks or about the impact Elephant will have on the way people use file systems.

Answering to these two important questions requires that people use the file system for many months. A long time period is required to determine whether versions deleted by Keep Landmarks or other policies will eventually be requested by a user or whether versions retained by such a policy will never be requested. Similarly, we believe that it takes time for users to gain sufficient confidence so that they feel free to modify or delete important files without first making backup copies. It is only when users have this confidence that we will gain insight into the intriguing question of how this new file-system model will change user behavior. Finally, long-term studies are needed to shed light on other issues related to user behavior such as possible heuristics for automatically assigning policies to files.

Our main contribution in this area, therefore, has been to enable future long-term studies by providing a working prototype and an experimentation framework. NFS shadowing will allow people to use the file system without having to trust it to store their data and our logging facility will record when they access old versions. In addition, the fact that the prototype allows users to define new policies should encourage users to experiment with a variety of policy ideas.

With these pieces in place, it is possible for us or other researchers to conduct the long-term user studies necessary to better understand the relationships between Elephant, its retention policies, and user behavior.

6.5 Summary

Our performance measurements show that there is no showstopper in the Elephant prototype. Performance is competitive with the standard FreeBSD FFS across a broad range of micro-benchmarks and some simple macro-benchmarks. We show that meta-data storage for versioned files can be 24 times larger than for non-versioned files, if a versioned file is stored in an inode log instead of an inode. This fact demonstrates the importance of our adaptive approach that uses inode logs only where necessary, to store files that currently have a history.

Our analysis of the file system traces and studies of the distribution of file types tell a common story. The majority of files, and specifically files that are written, are files that would normally not be versioned. The number of files for which versioning would be desirable comprise only a small fraction, approximately 12–15%, of typical file systems. This result is validated by the relatively small write rate we have calculated for the file system traces on versionable files. Given these results, we believe that the extra storage and disk write overhead incurred by using a file system such as Elephant is of minimal cost compared to the convenience and time gains (due to not having to restore or recreate accidentally-deleted files) made possible.

Finally, using NFS-server shadowing, we are able to exercise the prototype with a real user workload and we are beginning to gain some information from users regarding the usability of the system. Information about Keep Landmarks and other long-term retention policies, however, can only be obtained from a long-term user study, which is reserved for future work.

7 Related work

The goal of keeping multiple versions of data automatically, compactly, and in an organized way is reminiscent of software revision control systems [17, 24]. 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, while supporting the traditional file system interface, and thus freeing users from the details of version management.

The idea of versioned files was first proposed for the Cedar file system from Xerox PARC [20, 4]. 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 [1], and TOPS-10/-20 [13] 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 [6], Plan-9 [16, 15], and WAFL [5] 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 integrates tertiary storage with the file system and can thus retain all checkpoints, WAFL can keep as many as 20, 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 vulnerable as UFS users to losing all their morning's work in the afternoon, due to an inadvertent file deletion or overwrite.

The POSTGRES data base [23] maintains a complete history of database tables by archiving the transaction log. It also adds temporal operators to SQL to allow querying the state of the database at any point in the past. This provides database users with functionality similar to what Elephant gives file system users.

8 Other issues and future work

This section briefly discusses several remaining issues related to our design and prototype implementation that we are actively considering.

8.1 Version history export and import

Currently, Elephant stores files as version histories, but users can only access files one version at a time. This interface makes sense, because it matches the way that people access files in standard file systems. It is also useful, however, to allow users and applications to manipulate files at the granularity of their complete history. This facility is needed, for example, to move a file from one Elephant file system to another or to backup a file and its history.

We envision adding two new file-system operations to export and import file histories, either as kernel operations or as user-mode tools. Export would generate a single intermediate file containing the exported file's complete history. Import would reconstitute a versioned file from an intermediate file. Copies between Elephant file systems could be handled using a combined export-import operation, optimized to avoid creating the intermediate file.

An important open question is how to handle files that are managed by an application-defined policy. Ideally, the intermediate file would form a closure of both the exported file and its policy module. A file imported from another Elephant file system could then really be identical to the exported file.

8.2 Disconnected operation

Export and import operations allow users to use multiple Elephant systems to access a file, by copying it to the appropriate system before accessing it. It would be useful to extend this modest support to allow for full-fledged disconnected operation. In any file system, the key issue for disconnected operation is handling updates to a file that occur concurrently in multiple file systems [7]. For Elephant, the issue is somewhat more complicated, because it requires that an import operation be able to merge the version histories of multiple copies of a file. The key problem is that the merged history may have intervals in which multiple versions co-existed.

8.3 Version history merge and branch

Fundamental to Elephant's naming scheme is the assumption the multiple versions of a file never co-exist. We have just seen, however, that disconnected operation can violate this assumption. The same is true for software revision control systems.

A revision control system typically views file history as a rooted acyclic graph, allowing for branch and merge points. A branch occurs when a single version splits into two or more new versions of the file that can be modified independently. A merge combines multiple concurrent versions of a file into a single version. Merging thus allows branched versions to be reconciled periodically, before a software release, for example.

To support disconnected operation and applications such as RCS, we are exploring the idea of adding branch and merge points to Elephant file histories. The key issue is naming. Our current approach that uses time to name files must be augmented or replaced in order to resolve ambiguities that occur when multiple versions co-exist. One strategy is to add a tag to each concurrent history created by a branch. Versions could then be named by the pathname-time-tag triple when necessary to disambiguate among concurrent versions of a file.

9 Conclusions

Since their inception, file systems have contracted with their users to reliably store the most recent version of each file until that file is deleted. File systems have evolved excellent solutions to address a wide variety of network, system, and media failures. We believe that it is time to offer a richer contract in which the file system also protects users from their own mistakes. This has become feasible due to the recent arrival of very large cheap disk storage. Our analysis of file system trace data indicates that the amount of space required to provide this level of protection is moderate, on the order of 1.4 MBytes per day, per user.

Providing protection from user mistakes requires the separation of file system modification operations and file system storage reclamation. All operations in the file system that modify data must be revocable, meaning that copy-on-write techniques must be used to maintain all file system data, and no regular file system operation can free storage. Since file system modification has been separated from storage reclamation, we must define mechanisms and policies for storage reclamation. We have argued that the system must support the specification of storage reclamation policies at the granularity of individual files or groups of files. We have also described four storage reclamation policies that we believe will be valuable to users, and also how both these system-defined policies and application-defined policies can be implemented using a simple interface to the file versioning information maintained by the file system.

This paper has presented our arguments that this new contract between the file system and the user is desirable and feasible, and has described our initial attempt to build a file system, Elephant, which implements this new contract. Our experience with the system to date indicates that there is no substantial performance penalty for providing this additional level of protection.

Acknowledgments

We would like to thank Alex Brodsky, Yvonne Coady, Joon S. Ong, the anonymous referees, and our shepherd, Peter Chen, whose comments on earlier drafts of this paper have helped immensely, and Sreelatha Reddy who helped implement the NFS shadowing. We would also like to thank the students in the DSG lab and in CPSC 508 at UBC for agreeing to be guinea pigs for our Elephant user studies.

References

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

[2] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of SIGMETRICS '99*, pages 59–69, Atlanta, GA, USA, 1999.

[3] James Griffioen and Randy 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] R. Hagmann. Reimplementing the cedar file system using logging and group commit. *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, 21(5):155–162, November 1987.

[5] Dave Hitz, James Lau, and Michael 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.

[6] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[7] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Syste m Principles*, pages 213–225, Pacific Grove, CA, October 1991. ACM.

[8] K.Smith and M.Seltzer. A comparison of FFS disk allocation policies. In *Proc. 1996 USENIX Conference*, pages 15–26, January 1996.

[9] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptative methods. In *Proceedings of the 16th Symposium on Operating Systems Princi ples*, pages 238–252, 1997.

[10] M. McDonald and R. Bunt. Improving file system performance by dynamically restructuring disk space. In *Proc. of Phoenix Conference on Computers and Communication*, pages 264–269, March 1989.

[11] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[12] L. McVoy and S. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of the 1990 Summer Usenix*, pages 137–144, June 1990.

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

[14] J. K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, and J.G. Thompson. A trace-driven analysis of the UNIX 4.2BSD file system. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, WA, December 1985.

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

[16] Sean Quinlan. A cached worm file system. *Software—Practice and Experience*, 21(12):1289–1299, December 1991.

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

[18] Drew Roselli. Characteristics of file system workloads. Technical Report UCB//CSD-98-1029, Computer Science Division, University of California, Berkeley, 1998.

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

[20] Michael D. Schroeder, David K. Gifford, and Roger 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), December 1985. ACM.

[21] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering. In *Proc. 1995 Winter USENIX Conference*, pages 249–264, January 1995.

[22] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl 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.

[23] Michael Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 289–300, September 1987.

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