

Deciding When to Forget in the Elephant File System



Douglas. S. Santry, Michael J. Feeley, Norman
C. Hutchinson, Ross W. Carton, Jacob Ofir,
and Alistair. C. Veitch (1999)

Presented by:
Gary Huang & Nguyet Nguyen

November 20, 2006



Outline

- Motivation
- Key Issues
- Design
- Implementation
- Performance
- Conclusion
- Discussion



Motivation

- **Need:** FS protects users from their mistakes
- Existing FSs
 - Cedar – protection from accidental overwrite, but not delete
 - Trash can – protection from delete but not overwrite. Also provide limited undo capacity
 - Checkpoint – change btw checkpoints not recoverable and limited number and frequency of checkpoints.
- Solutions from users and application
 - Users make and maintain multiple copies of data and avoid deletes whenever possible.
 - File editors provide “undo” tool.



Key Issues

- **Store reclamation** separated from **file write** and **read**
 - Deleting a file must not release its storage and file updates must not overwrite existing file data.
- **Variety of retention policies**
 - Many different types of files (Read-only, Derived, Cashed, Temporary, user-modified)
 - Specified by users, BUT implemented by the system
- **Undo**
 - Require complete history for limited period of time
- **Long term histories**
 - NOT retain all versions
 - FS assists the user in deciding what versions

ELEPHANT file system

-deciding when to forget



Design

Implementation

Performance



Design: Key Principle

- Separate *storage management* from the common file system *operations*
- Deleting a file does not release its storage
- **Copy-on-write** fashion
 - creating a new version of a file block when it is written



Design: File retention policies

- Keep One – No versioning
- Keep All – Complete versioning
- Keep Safe – Undo protection
- Keep Landmarks – Long-term history



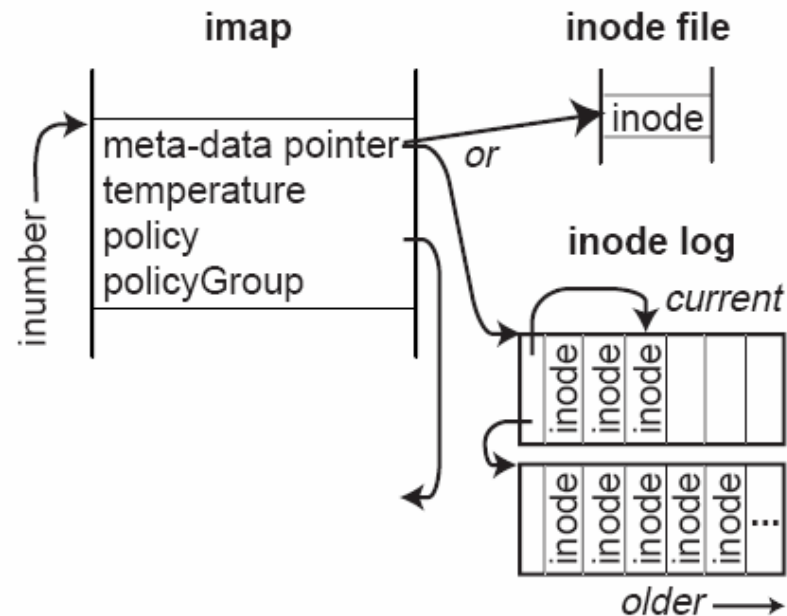
Implementation

- Defining Versions
 - Inode
 - Inode log
 - Name log
 - Nonversioned inodes

Implementation

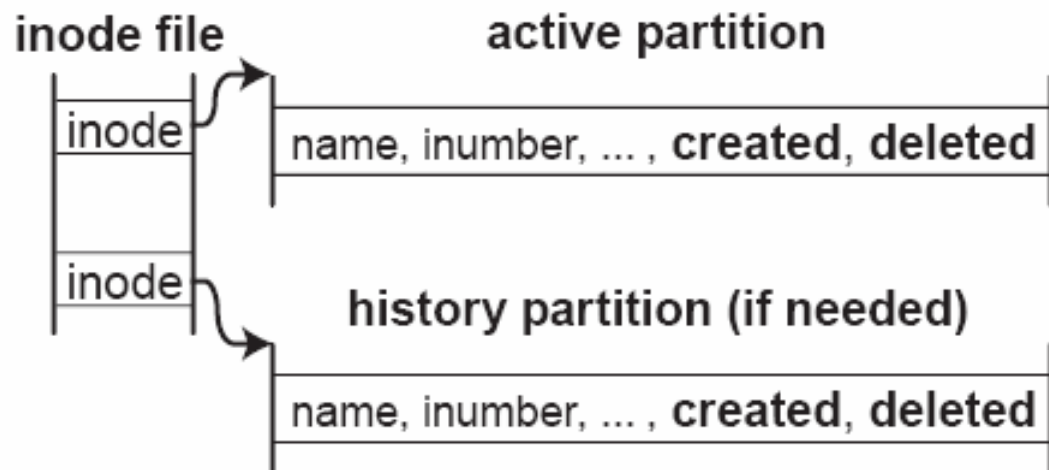
- Imap

provides a level of indirection between an inumber and the disk address of a file's inode or inode log.



Implementation

- Directories
 - Initially, one inode (called active inode)
 - After deletes, a second inode
 - Scan both inodes





Implementation

- System Interface for User-Mode App
 - setCurrentEpoch (timestamp)
 - setHistory (file) => history
 - setLandmark (file)
 - unsetLandmark (file)
 - setPolicy (file, policyID)
 - groupFiles (fileA, FileB)
 - ungroupFile (file)



Implementation

- System Interface for Storage Reclamation
 - mapImap (pathN) => mntPt
 - lockFile (mntPt, iNum)
 - unlockFile (mntPt, iNum)
 - readBlock (mntPt, block)
 - writeBlock (mntPt, block)
 - freeBlock (mntPt, block)



Implementation

- System Interface for Application defined Retention Policies
 - registerPolicy (pathN) => policyID
 - unregisterPolicy (policyID)
 - lookupPolicy (policyID) => pathN
 - cleanFile (fileHistory) => (verList, newTemp)



Performance

- Compare the performance of our Elephant prototype to the standard FreeBSD FFS file system
- Examine the types of files stored by a large file system to estimate what portion of its files would be versioned
- Analyze file-system trace data to estimate how much extra storage an Elephant file system might consume for history information



Performance

Elephant vs. FFS

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



Performance

File system profile

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

- **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



Performance

Extra Storage Consumed

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



Conclusion

- Providing protection from user mistakes requires the separation of file system modification operations and file system storage reclamation.
- Four storage reclamation policies that are valuable to users
- Both system defined policies and application-defined policies can be implemented using a simple interface for file versioning

DISCUSSION:

Deciding when to forget in the
Elephant file system

- Should we need such strong protection from the file system to protect user from their mistakes?
- What do you think about the claim:
“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”

Do you think it is a waste?

About related work

- Is it worth using the Elephant File System while we have ...
 - Version control systems. E.g. CVS, time machine in MacOS X
 - Backup systems (daily, weekly, and monthly). E.g. checkpoint FS
 - Or even Window's recycle bin

About Implementation

- Should the file size be considered in deciding to keep versions? Is it a problem when versioning large files (e.g. movie, music, image,...)?
- Do you think the data structures (inode-log, name logs, imap,...) are efficient? And the overall design and implementation? (Consider: performance,...)

Security

- Should the file system have some support to protect against the attacks like some trojan or malicious programs?
 - You are assuming that your file is on “Keep Landmarks”, BUT something purges all your old versions and turns it to “Keep One”
 - Should we have a system recovery tool to go along with Elephant? Or versions backup utility?

About applicability

- Why isn't the Elephant File System widely used today? Are people reluctant to using unfamiliar file systems?