PersiFS

A Versioned File System with an Efficient Representation

Dan R. K. Ports, Austin T. Clements, and Erik D. Demaine
MIT Computer Science and Artificial Intelligence Laboratory
{drkp,aclements,edemaine}@mit.edu

Introduction

Most file systems are *ephemeral*, meaning that once a change has been made, there is no way to recall the previous contents of the file system. **PersiFS** is a *continuously versioned* file system. It stores every modification to the file system, allowing access to *any* past version of any file. PersiFS achieves this without sacrificing access time to either current versions or past versions, using inordinate amounts of disk space, or requiring modification to existing applications.

In order to make continuous versioning feasible, PersiFS tackles the problems of time and space efficiency with a number of efficient data structures for indexing and retaining files.

Advantages of PersiFS

- **Continuous versioning** — PersiFS retains *every* modification to the file system, allowing users to examine any file as it was at any point in time.
- **Efficient access to past revisions** — Operations on the current revision are only slightly slower than operations in normal B$^+$-tree-based file systems. PersiFS, however, also provides the *same time guarantees* for accessing *past* revisions. Reading from any revision or modifying the current revision requires disk accesses only logarithmic in the size of that revision, and committing the current revision does not require any disk accesses.
- **Efficient storage of past revisions** — PersiFS combines content addressable storage with content-sensitive chunking to efficiently coalesce portions of files on disk. This is particularly effective for sharing data between incremental revisions of the same file, though the same mechanism can share data between any revisions of any files.
- **Direct access to past revisions** — PersiFS directly exposes access to past file revisions through the file system interface, allowing convenient access with standard file system tools. The current version of the file system tree is available read-write at `/persifs/now`, and read-only views of previous versions are automatically mounted simply by specifying a timestamp instead of "`now`". No special tools or per-file version numbers are required, though convenience tools exist, e.g. for listing all revisions of a particular file.
- **Homogeneous representation** — Both the current revision and past revisions are stored in the *same data structures*. Unlike logging-based systems and some snapshot-based systems, there is no need to maintain a separate copy of the current version of the file system. The same mechanisms provide efficient access to both the current revision and past revisions.

Related Work

Snapshotting

Previous states of the file system are commonly stored by a backup system that periodically archives the state of the filesystem. *Snapshot*-based version-controlled file systems are the natural extension of this idea: they periodically take a snapshot of the state of the entire filesystem, and make it readily available.

File systems that use snapshot-based versioning include **Plan 9**, which stores daily snapshots of the file system and stores them on a content-addressable Venti archival server [PPD$^+$95], and **WAFL**, which uses copy-on-write disk blocks to save space [HLM94]. **AFS** also uses copy-on-write files to provide access to the most recent snapshot through the `OldFiles` directory

While snapshotting systems are typically space efficient, they are incapable of tracking changes that occur between snapshots. Furthermore, they have inflexible policy because snapshots are taken of the entire file system at once. PersiFS uses continuous versioning, which does not suffer from these problems.

Logging

Continuously versioned file systems are usually implemented using an undo or redo log. This allows any version to be recovered, but requires replaying the log, a potentially expensive operation. The disk access cost is asymptotically linear in the size of the log, so typically some form of periodic log checkpointing is necessary to circumvent the need to replay the log from the beginning.

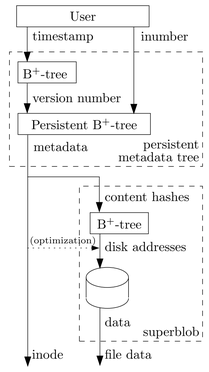
Because replaying a log is inefficient, the log is generally combined with a separate copy of the current revision in a structure such as a B$^+$-tree. This optimization is unnecessary in PersiFS, since its representation is approximately as fast as unversioned B$^+$-tree-based file systems when operating on the current revision. Moreover, it can retrieve previous revisions with the same efficiency.

File systems that use logging to implement continuous versioning include **VersionFS** [MR03] and **Wayback** [CDB04]. **CVFS** combines logging with a standard B-tree containing multiple versions [SGSG03], an approach that does not match the asymptotic guarantees of PersiFS.

PersiFS Structures

PersiFS stores its data directly on disk using two main data structures:

- The **persistent metadata tree** contains the history of the file system. It can efficiently obtain either past or present metadata for a specific file, regardless of the number of revisions in the system.
- The **superblob** stores file content. It reduces storage costs by exploiting similarity between file revision.



Persistent Metadata Tree

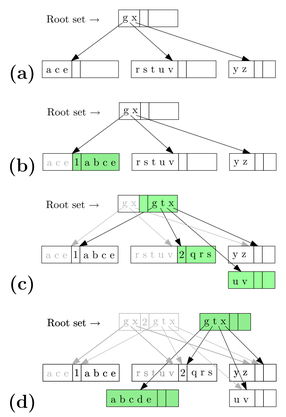
A versioned file system must be able to recover the previous state of any file, without sacrificing the speed of accessing or modifying the current revision. Most other continuously-versioned file systems require an expensive log replay to access previous revisions, and keep a separate copy of the current revision in a more efficient structure. Instead, the **persistent metadata tree** at the core of PersiFS makes accessing past revisions as fast as accessing the current one.

To achieve this, PersiFS uses ***partially persistent* data structures**, which are data structures that can answer queries about any of their previous states. Both past and present metadata are stored in a **partially persistent B$^+$-tree**. This data structure provides the following asymptotic property:

Theorem. *Queries or modifications on a partially persistent B^+-tree can be performed using $O(\log_{B+1} N)$ disk accesses in the worst case, where N is the number of files in the affected revision, and B the block size.*

This logarithmic cost is considerably faster than the linear cost required to scan a log. It is also a considerable improvement over a standard tree: the persistent B$^+$-tree's access cost is logarithmic in the number of files *in the affected revision*, while a standard tree would incur costs logarithmic in the size of *the complete history of the file system*.

To implement our partially-persistent B$^+$-tree, we use a variant on a well-known technique [DSST89]. We add a *modification box* to each internal node of the B$^+$-tree. This consists of a second copy of the contents of the node and a timestamp, allowing a modification to be recorded along with the time it took effect.
When a node in the tree needs to be modified, the changed version is stored in the modification box if it is free. If it is full, a new copy of the node is made, and the parent node is modified to point to the new node. Modifying the parent may require a cascading chain of copies, but an argument based on amortized analysis gives the desired logarithmic bound. Thus, the cost due to persistence is only a small constant factor greater than a single-version tree.
Performing a search simply requires checking the time stamp at each node traversed in order to decide whether to use or ignore the change stored in the modification box, which adds little additional cost.



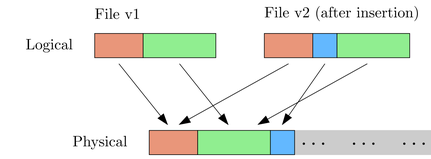
Four modifications to a persistent B-tree. Data not used in the latest revision is shown in gray; changes made by the latest modification are shown in green.
(a) The initial tree at revision 0.
(b) Inserting b changes a modification box and creates revision 1.
(c) Inserting q requires a split and creates revision 2.
(d) Inserting d requires creating a new node for revision 3 because the modification box is already full.

Superblob

File data is not stored directly in the persistent metadata tree, but rather in the **superblob**. The goal of this indirection layer is to minimize the amount of storage space required. There is a great deal of similarity between files in the file system. Many changes affect only a small part of a file, creating a new revision largely similar to the previous one; moreover, different files may have similar content, such as when a file is copied. The superblob attempts to store identical content on disk only once, regardless of how many files or revisions it appears in.

To achieve this goal, files are divided into **chunks**, which are stored in the superblob. Rather than place chunk boundaries at regular intervals, we use **content-sensitive chunking**. This technique places chunk boundaries based on file contents such that local modifications to file contents, including insertions and deletions, only affect chunks in that region of the file. We use the same Rabin fingerprint-based algorithm as LBFS [MCM01].

Chunks are identified by SHA-1 hashes of their contents. Thus, the superblob is a **content-addressable** storage system similar to Venti [QD02]. The file metadata contains a list of the chunks that comprise the file. The superblob is indexed by a B$^+$-tree, which makes it possible to determine whether a particular chunk has already been stored — if so, it can be reused. As an optimization, the list of chunks in the file metadata can be represented as disk addresses, so the superblob index B$^+$-tree needs only be consulted on writes.



Two file versions sharing identical chunks in the superblob. The new version adds a third chunk between the existing two. The superblob index B$^+$-tree maps the chunk hashes for the existing chunks to their data locations on disk.

Future Work

- **Retention policy** — Versioned file systems necessarily require large volumes of storage, but by allowing users and administrators to set policies on what is retained, this space requirement can be significantly reduced.
  - **Expiration policy** — Support for flexible policies that allow the administrator to specify when old revisions are removed, in order to reduce storage usage. In addition to expiring old revisions, intermediate revisions could be removed, causing the system to gracefully shift from continuous versioning to snapshotting for times in the distant past.
  - **Exclusion policy** — Support for policy regarding what types of files to exclude from continuous versioning. For some files, such as temporary files and intermediate files, versioning is of questionable utility. Users should be able to express this in order to save space.
- **Locality optimizations** — Rearranging file chunks on disk for speed. Storing data only once in the superblob is a trade-off between clustering of a file's chunks and storage space savings, but relocating or replicating some chunks can increase locality for common cases.

PersiFS: Bringing the Past to the Present for a Better Future

References

[CDB04] B. Cornell, P. Dinda, and F. Bustamante, *Wayback: A user-level versioning file system for Linux*, Proc. USENIX '04 (Boston, MA), June 2004.

[DSST89] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, *Making data structures persistent*, Journal of Computer and System Sciences **38** (1989), 86–124.

[HLM94] D. Hitz, J. Lau, and M. Malcolm, *File system design for an NFS file server appliance*, Proc. USENIX '94 (San Francisco, CA), 17–21 1994, pp. 235–246.

[MCM01] A. Muthitacharoen, B. Chen, and D. Mazieres, *A low-bandwidth network file system*, Proc. SOSP '01, 2001, pp. 174–187.

[MR03] K-K. Muniswamy-Reddy, *VERSIONFS: A versatile and user-oriented versioning file system*, Master's thesis, Stony Brook University, December 2003.

[PPD$^+$95] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, *Plan 9 from Bell Labs*, Computing Systems **8** (1995), no. 3, 221–254.

[QD02] S. Quinlan and S. Dorward, *Venti: a new approach to archival storage*, Proc. FAST '02 (Monterey, CA), 2002.

[SGSG03] C. Soules, G. Goodson, J. Strunk, and G. Ganger, *Metadata efficiency in versioning file systems*, Proc. FAST '03 (San Francisco, CA), March 2003.