

# PersiFS: A Versioned File System with an Efficient Representation

Dan R. K. Ports, Austin T. Clements, and Erik D. Demaine \*

The availability of previous file versions is invaluable for recovering from file corruption or user errors such as accidental deletions. *Versioned* file systems address this need by retaining earlier versions of changed files. Many existing file systems, such as Plan 9, WAFL, AFS, and others, use a *snapshotting* approach: they record and archive the state of the file system at periodic intervals. However, this fails to capture modifications that are made between snapshots. Our system, PersiFS, is *continuously versioned*, meaning that it stores every modification, and thus allows access to the file system state as it appeared at any specified time. To make this feasible, we use a number of efficient data structures to optimize both access time and disk space.

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.

Efficiently storing all previous revisions of each file presents a new set of challenges because of the volume of data involved. Our major contribution is a compact representation that supports efficient queries and modifications. Most previous continuously versioned systems, such as CVFS, VersionFS, and Wayback, use a log-based representation that requires an expensive scan of the log to access earlier revisions, and periodic snapshots. Instead, PersiFS uses *partially persistent* data structures, which are data structures that can answer queries about any of their previous states.

PersiFS stores all past and present file metadata in a *partially persistent B<sup>+</sup>-tree*, eliminating the need for logs and snapshots. Our partially persistent B<sup>+</sup>-tree provides read-write access to the current revision and read-only access to previous revisions

with theoretical guarantees on worst-case performance. 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. Thus, accesses and modifications require approximately as little time as standard, unversioned B<sup>+</sup>-tree-based file systems take to operate on the current revision. Moreover, unlike logging designs, PersiFS can access *any* previous revision with the same efficiency, and there is no need to maintain a separate copy of the current revision.

Since disk sizes are not yet infinite, PersiFS exploits the similarity between files in different revisions in order to minimize the amount of storage space required. This is achieved with indirect storage: the B<sup>+</sup>-tree contains only metadata and pointers to file contents stored in a separate structure called the *superblob*. File contents are divided into *chunks* using the LBFS variable-sized segmentation algorithm, which places chunk boundaries based on content rather than at fixed offsets. Like Venti, PersiFS identifies chunks by a hash of their contents, so it does not need to store identical chunks more than once. The result is that PersiFS can share identical content on disk between any revision of any file, thus greatly reducing file storage costs.

Our early prototype of PersiFS includes both the representation described above and a standard log-based backend. Comparison measurements indicate that PersiFS’s representation yields substantially better performance than logging for many workloads, as well as significant improvements in storage costs, without sacrificing the speed of regular access. This system also provides worst-case guarantees per operation, whereas snapshotting generally significantly slows down system performance during snapshot operations.

Current work focuses on refining our implementation and optimizing performance. In particular, we are researching techniques for rearranging file chunks on disk to exploit locality. We are also adding support for flexible retention and expiration policies to reduce storage usage.

---

\*MIT Computer Science and Artificial Intelligence Laboratory; {drkp, aclements, edemaine}@mit.edu