2nd USENIX Conference on File and Storage Technologies, San Francisco, CA, Mar 31 - Apr 2, 2003.

Metadata Efficiency in Versioning File Systems

Craig A.N. Soules, Garth R. Goodson, John D. Strunk, Gregory R. Ganger
Carnegie Mellon University

Abstract

Versioning file systems retain earlier versions of modified files, allowing recovery from user mistakes or system corruption. Unfortunately, conventional versioning systems do not efficiently record large numbers of versions. In particular, versioned metadata can consume as much space as versioned data. This paper examines two space-efficient metadata structures for versioning file systems and describes their integration into the Comprehensive Versioning File System (CVFS), which keeps all versions of all files. *Journal-based metadata* encodes each metadata version into a single journal entry; CVFS uses this structure for inodes and indirect blocks, reducing the associated space requirements by 80%. *Multiversion b-trees* extend each entry's key with a timestamp and keep current and historical entries in a single tree; CVFS uses this structure for directories, reducing the associated space requirements by 99%. Similar space reductions are predicted via trace analysis for other versioning strategies (e.g., on-close versioning). Experiments with CVFS verify that its current-version performance is similar to that of non-versioning file systems while reducing overall space needed for history data by a factor of two. Although access to historical versions is slower than conventional versioning systems, checkpointing is shown to mitigate and bound this effect.

1 Introduction

Self-securing storage [41] is a new use for versioning in which storage servers internally retain file versions to provide detailed information for post-intrusion diagnosis and recovery of compromised client systems [40]. We envision self-securing storage servers that retain every version of every file, where every modification (e.g., a WRITE operation or an attribute change) creates a new version. Such *comprehensive versioning* maximizes the information available for post-intrusion diagnosis. Specifically, it avoids pruning away file versions, since this might obscure intruder actions. For self-securing storage, *pruning techniques* are particularly dangerous when they rely on client-provided information, such as CLOSE operations — the versioning is being done specifically to protect stored data from malicious clients.

Obviously, finite storage capacities will limit the duration of time over which comprehensive versioning is possible. To be effective for intrusion diagnosis and recovery, this duration must be greater than the intrusion detection latency (i.e., the time from an intrusion to when it is detected). We refer to the desired duration as the *detection window*. In practice, the duration is limited by the rate of data change and the space efficiency of the versioning system. The rate of data change is an inherent aspect of a given environment, and an analysis of several real environments suggests that detection windows of several weeks or more can be achieved with only a 20% cost in storage capacity [41].

In a previous paper [41], we described a prototype self-securing storage system. By using standard copy-on-write and a log-structured data organization, the prototype provided comprehensive versioning with minimal performance overhead ($<$10%) and reasonable space efficiency. In that work, we discovered that a key design requirement is efficient encoding of metadata versions (the additional information required to track the data versions). While copy-on-write reduces data versioning costs, conventional versioning implementations still involve one or more new metadata blocks per version. On average, the metadata versions require as much space as the versioned data, halving the achievable detection window. Even with less comprehensive versioning, such as Elephant [37] or VMS [29], the metadata history can become almost ($\approx$80%) as large as the data history.

This paper describes and evaluates two methods of storing metadata versions more compactly: journal-based metadata and multiversion b-trees. Journal-based metadata encodes each version of a file's metadata in a journal entry. Each entry describes the difference between two versions, allowing the system to roll-back to the earlier version of the metadata. Multiversion b-trees retain all versions of a metadata structure within a single tree. Each entry in the tree is marked with a timestamp indicating the time over which the entry is valid.

The two mechanisms have different strengths and weaknesses. We discuss these and describe how both techniques are integrated into a comprehensive versioning file system called CVFS. CVFS uses journal-based metadata for inodes and indirect blocks to encode changes to attributes and file data pointers; doing so reduces the space used for their histories by 80%. CVFS implements directories as multiversion b-trees to encode additions and removals of directory entries; doing so reduces the space used for their histories by 99%. Combined, these mechanisms nearly double the potential detection window over conventional versioning mechanisms, without increasing the access time to current versions of the data.

Journal-based metadata and multiversion b-trees are also valuable for conventional versioning systems. Using these mechanisms with on-close versioning and snapshots would provide similar reductions in versioned metadata. For on-close versioning, this reduces the total space required by nearly 35%, thereby reducing the pressure to prune version histories. Identifying solid heuristics for such pruning remains an open area of research [37], and less pruning means fewer opportunities to mistakenly prune important versions.

The rest of this paper is divided as follows. Section 2 discusses conventional versioning and motivates this work. Section 3 discusses the two space-efficient metadata versioning mechanisms and their tradeoffs. Section 4 describes the CVFS versioning file system. Section 5 analyzes the efficiency of CVFS in terms of space efficiency and performance. Section 6 describes how our versioning techniques could be applied to other systems. Section 7 discusses additional related work. Section 8 summarizes the paper's contributions.

2 Versioning and Space Efficiency

Every modification to a file inherently results in a new version of the file. Instead of replacing the previous version with the new one, a *versioning file system* retains both. Users of such a system can then access any historical versions that the system keeps as well as the most recent one. This section discusses uses of versioning, techniques for managing the associated capacity costs, and our goal of minimizing the metadata required to track file versions.

2.1 Uses of Versioning

File versioning offers several benefits to both users and system administrators. These benefits can be grouped into three categories: recovery from user mistakes, recovery from system corruption, and analysis of historical changes. Each category stresses different features of the versioning system beneath it.

Recovery from user mistakes: Human users make mistakes, such as deleting or erroneously modifying files. Versioning can help [17, 29, 37]. Recovery from such mistakes usually starts with some a priori knowledge about the nature of the mistake. Often, the exact file that should be recovered is known. Additionally, there are only certain versions that are of any value to the user; intermediate versions that contain incomplete data are useless. Therefore, versioning aimed at recovery from user mistakes should focus on retaining key versions of important files.

Recovery from system corruption: When a system becomes corrupted, administrators generally have no knowledge about the scope of the damage. Because of this, they restore the entire state of the file system from some well-known "good" time. A common versioning technique to help with this is the online *snapshot*. Like a backup, a snapshot contains a version of every file in the system at a particular time. Thus, snapshot systems present sets of known-valid system images at a set of well-known times.

Analysis of historical changes: A history of versions can help answer questions about how a file reached a certain state. For example, version control systems (e.g., RCS [43], CVS [16]) keep a complete record of committed changes to specific files. In addition to selective recovery, this record allows developers to figure out who made specific changes and when those changes were made. Similarly, self-securing storage seeks to enable post-intrusion diagnosis by providing a record of what happened to stored files before, during, and after an intrusion. We believe that every version of every file should be kept. Otherwise, intruders who learn the pruning heuristic will leverage this information to prune any file versions that might disclose their activities. For example, intruders may make changes and then quickly revert them once damage is caused in order to hide their tracks. With a complete history, administrators can determine which files were changed and estimate damage. Further, they can answer (or at least construct informed hypotheses for) questions such as "When and how did the intruder get in?" and "What was their goal?" [40].

2.2 Pruning Heuristics

A true comprehensive versioning system keeps all versions of all files for all time. Such a system could support all three goals described above. Unfortunately, storing this much information is not practical. As a result, all versioning systems use *pruning heuristics*. These pruning heuristics determine when versions should be created and when they should be removed. In other words, pruning heuristics determine which versions to keep from the total set of versions that would be available in a comprehensive versioning system.

2.2.1 Common Heuristics

A common pruning technique in versioning file systems is *on-close* versioning. This technique keeps only the last version of a file from each session; that is, each CLOSE of a file creates a distinct version. For example, the VMS file system [29] retains a fixed number of versions for each file. VMS's pruning heuristic creates a version after each CLOSE of a file and, if the file already has the maximum number of versions, removes the oldest remaining version of the file. The more recent Elephant file system [37] also creates new versions after each CLOSE; however, it makes additional pruning decisions based on a set of rules derived from observed user behavior.

Version control systems prune in two ways. First, they retain only those versions explicitly committed by a user. Second, they retain versions for only an explicitly-chosen subset of the files on a system.

By design, snapshot systems like WAFL [19] and Venti/Plan9 [34] prune all of the versions of files that are made between snapshots. Generally, these systems only create and delete snapshots on request, meaning that the system's administrator decides most aspects of the pruning heuristic.

2.2.2 Information Loss

Pruning heuristics act as a form of lossy compression. Rather than storing every version of a file, these heuristics throw some data away to save space. The result is that, just as a JPEG file loses some of its visual clarity with lossy compression, pruning heuristics reduce the clarity of the actions that were performed on the file.

Although this loss of information could result in annoyances for users and administrators attempting to recover data, the real problem arises when versioning is used to analyze historical changes. When versioning for intrusion survival, as in the case of self-securing storage, pruning heuristics create holes in the administrator's view of the system. Even creating a version on every CLOSE is not enough, as malicious users can leverage this heuristic to hide their actions (e.g., storing exploit tools in an open file and then truncating the file to zero before closing it).

To avoid traditional pruning heuristics, self-securing storage employs comprehensive versioning over a fixed window of time, expiring versions once they become older than the given window. This detection window can be thought of as the amount of time that an administrator has to detect, diagnose, and recover from an intrusion. As long as an intrusion is detected within the window, the administrator has access to the entire sequence of modifications since the intrusion.

2.3 Lossless Version Compression

For a system to avoid pruning heuristics, even over a fixed window of time, it needs some form of lossless version compression. Lossless version compression can also be combined with pruning heuristics to provide further space reductions in conventional systems. To maximize the benefits, a system must attempt to compress both versioned data and versioned metadata.

Data: Data block sharing is a common form of lossless compression in versioning systems. Unchanged data blocks are shared between versions by having their individual metadata point to the same physical block. Copy-on-write is used to avoid corrupting old versions if the block is modified.

An improvement on block sharing is byte-range differencing between versions. Rather than keeping the data blocks that have changed, the system keeps the bytes that have changed [27]. This is especially useful in situations where a small change is made to the file. For example, if a single byte is inserted at the beginning of a file, a block sharing system keeps two full copies of the entire file (since the data of every block in the file is shifted forward by one byte); for the same scenario, a differencing system only stores the single byte that was added and a small description of the change.

Another recent improvement in data compression is hash-based data storage [31, 34]. These methods recognize identical blocks or ranges of data across the system and store only one copy of the data. This method is quite effective for snapshot versioning systems, and could likely be applied to other versioning systems with similar results.

Metadata: Conventional versioning file systems keep a full copy of the file metadata with each version. While it simplifies version access, this method quickly exhausts capacity, since even small changes to file data or attributes result in a new copy of the metadata.

Figure 1 shows an example of how the space overhead of versioned metadata can become a problem in a conventional versioning system. In this example, a program is writing small log entries to the end of a large file. Since several log entries fit within a single data block, appending entries to the end of the file produces several different versions of the same block. Because each versioned data block has a different location on disk, the system must create a new version of the indirect block to track its location. In addition, the system must write a new version of the inode to track the location of the versioned indirect block. Since any data or metadata change will always result in a new version of the inode, each version is tracked using a pointer to that version's inode. Thus, writing a

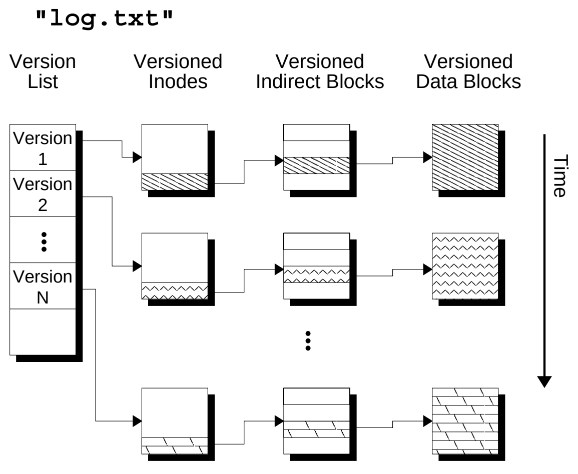


Figure 1: **Conventional versioning system.** In this example, a single logical block of file "log.txt" is overwritten several times. With each new version of the data block, new versions of the indirect block and inode that reference it are created. Notice that although only a single pointer has changed in both the indirect block and the inode, they must be rewritten entirely, since they require new versions. The system tracks each version with a pointer to that version's inode.

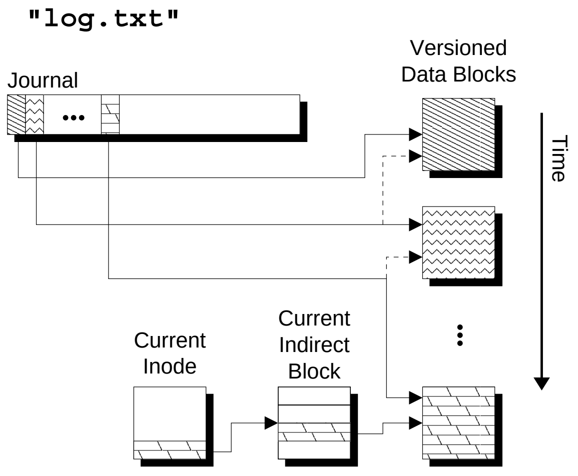


Figure 2: **Journal-based metadata system.** Just as in Figure 1, this figure shows a single logical block of file "log.txt" being overwritten several times. Journal-based metadata retains all versions of the data block by recording each in a journal entry. Each entry points to both the new block and the block that was overwritten. Only the current version of the inode and indirect block are kept, significantly reducing the amount of space required for metadata.

single data block results in a new indirect block, a new inode, and an entry in the version list, resulting in more metadata being written than data.

Access patterns that create such metadata versioning problems are common. Many applications create or modify files piece by piece. In addition, distributed file systems such as NFS create this behavior by breaking large updates of a file into separate, block-sized updates. Since there is no way for the server to determine if these block-sized writes are one large update or several small ones, each must be treated as a separate update, resulting in several new versions of the file.

Again, the solution to this problem is some form of differencing between the versions. Mechanisms for creating and storing differences of metadata versions are the main focus of this work.

2.4 Objective

In a perfect world we could keep all versions of all files for an infinite amount of time with no impact on performance. This is obviously not possible. The objective of this work is to minimize the space overhead of versioned metadata. For self-securing storage, doing so will increase the detection window. For other versioning purposes, doing so will reduce the pressure to prune. Because this space reduction will require compressing metadata versions, it is also important that the performance overhead of both version creation and version access be minimized.

3 Efficient Metadata Versioning

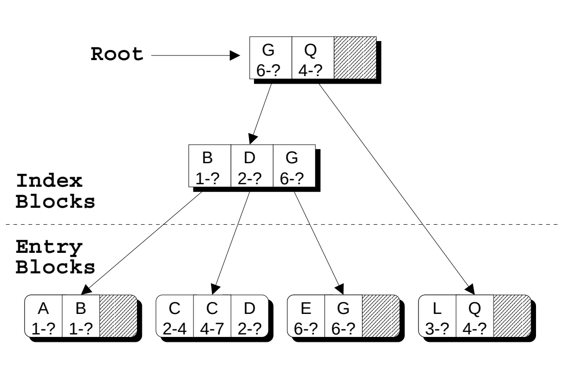
One characteristic of versioned metadata is that the actual changes to the metadata between versions are generally quite small. In Figure 1, although an inode and an indirect block are written with each new version of the file, the only change to the metadata is an update to a single block pointer. The system can leverage these small changes to provide much more space-efficient metadata versioning. This section describes two methods that leverage small metadata modifications, and Section 4 describes an implementation of these solutions.

3.1 Journal-based Metadata

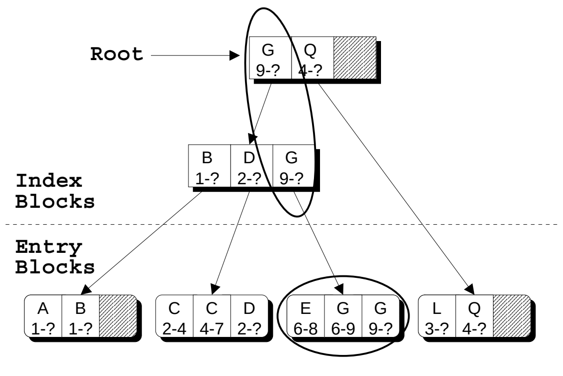
Journal-based metadata maintains a full copy of the current version's metadata and a journal of each previous metadata change. To recreate old versions of the metadata, each change is undone backward through the journal until the desired version is recreated. This process of undoing metadata changes is referred to as *journal rollback*.

Figure 2 illustrates how journal-based metadata works in the example of writing log entries. Just as in Figure 1, the system writes a new data block for each version; however, in journal-based metadata, these blocks are tracked using small journal entries that track the locations of the new and old blocks. By keeping the current version of the metadata up-to-date, the journal entries can be rolled-back to any previous version of the file.

In addition to storing version information, the journal can



(a) Initial tree structure.



(b) After removal of E and update of G.

Figure 3: **Multiversion b-tree.** This figure shows the layout of a multiversion b-tree. Each entry of the tree is designated by a ⟨user-key, timestamp⟩ tuple which acts as a key for the entry. A question mark (?) in the timestamp indicates that the entry is valid through the current time. Different versions of an entry are separate entries using the same user-key with different timestamps. Entries are packed into entry blocks, which are tracked using index blocks. Each index pointer holds the key of the last entry along the subtree that it points to.

be used as a write-ahead log for metadata consistency, just as in a conventional journaling file system. To do so, the new block pointer must be recorded in addition to the old. Using this, a journal-based metadata implementation can safely maintain the current version of the metadata in memory, flushing it to disk only when it is forced from the cache.

3.1.1 Space vs. Performance

Journal-based metadata is more space efficient than conventional versioning. However, it must pay a performance penalty for recreating old versions of the metadata. Since each entry written between the current version and the requested version must be read and rolled-back, there is a linear relation between the number of changes to a file and the performance penalty for recreating old versions.

One way the system can reduce this overhead is to *checkpoint* a full copy of a file's metadata to the disk occasionally. By storing checkpoints and remembering their locations, a system can start journal roll-back from the closest checkpoint in time rather than always starting with the current version. The frequency with which these checkpoints are written dictates the space/performance trade-off. If the system keeps a checkpoint with each modification, journal-based metadata performs like a conventional versioning scheme (using the most space, but offering the best back-in-time performance). However, if no checkpoints are written, the only full instance of the metadata is the current version, resulting in the lowest space utilization but reduced back-in-time performance.

3.2 Multiversion B-trees

A multiversion b-tree is a variation on standard b-trees that keeps old versions of entries in the tree [2]. As in a standard b-tree, an entry in a multiversion b-tree contains a key/data pair; however, the key consists of both a user-defined key and the time at which the entry was written. With the addition of this time-stamp, the key for each version of an entry becomes unique. Having unique keys means that entries within the tree are never overwritten; therefore, multiversion b-trees can have the same basic structure and operations as a standard b-tree. To facilitate current version lookups, entries are sorted first by the user-defined key and then by the timestamp.

Figure 3a shows an example of a multiversion b-tree. Each entry contains both the user-defined key and the time over which the entry is valid. The entries are packed into entry blocks, which act as the leaf nodes of the tree. The entry blocks are tracked using index blocks, just as in standard b+trees. In this example, each pointer in the index block references the last entry of the subtree beneath it. So in the case of the root block, the G subtree holds all entries with values less than or equal to G, with $\langle G, 6{-}?\rangle$ as its last entry. The Q subtree holds all entries with values between G and Q, with $\langle Q, 4{-}?\rangle$ as its last entry.

Figure 3b shows the tree after a remove of entry E and an update to entry G. When entry E is removed at time 8, the only change is an update to the entry's timestamp. This indicates that E is only valid from time 6 through time 8. When entry G is updated at time 9, a new entry is created and associated with the new data. Also, the old entry for G must be updated to indicate its bounded window of validity. In this case, the index blocks must also be updated to reflect the new state of the subtree, since the last entry of the subtree has changed.

Since both current and history entries are stored in the same tree, accesses to old and current versions have the same performance. For this reason, large numbers of history entries can decrease the performance of accessing current entries.

3.3 Solution Comparison

Both journal-based metadata and multiversion b-trees reduce the space utilization of versioning but incur some performance penalty. Journal-based metadata pays with reduced back-in-time performance. Multiversion b-trees pay with reduced current version performance.

Because the two mechanisms have different drawbacks, they each perform certain operations more efficiently. As mentioned above, the number of history entries in a multiversion b-tree can adversely affect the performance of accessing the current version. This emerges in two situations: linear scan operations and files with a large number of versions. The penalty on lookup operations is reduced by the logarithmic nature of the tree structure, but large numbers of history entries can increase tree depth. Linear scanning of all current entries requires accessing every entry in the tree, which becomes expensive if the number of history entries is high. In both of these cases, it is better to use journal-based metadata.

When lookup of a single entry is common or history access time is important, it is preferable to use multiversion b-trees. Using a multiversion b-tree, all versions of the entry are located together in the tree and have logarithmic lookup time (for both current and history entries), giving a performance benefit over the linear roll-back operation required by journal-based metadata.

4 Implementation

We have integrated journal-based metadata and multiversion b-trees into a comprehensive versioning file system, called CVFS. CVFS provides comprehensive versioning within our self-securing NFS server prototype. Because of this, some of our design decisions (such as the implementation of a strict detection window) are specific to self-securing storage. Regardless, these structures would be effective in most versioning systems.

4.1 Overview

Since current versions of file data must not be overwritten in a comprehensive versioning system, CVFS uses a log-structured data layout similar to LFS [36]. Not only does this eliminate overwriting of old versions on disk, but it also improves update performance by combining data and metadata updates into a single disk write.

CVFS uses both mechanisms described in Section 3. It uses journal-based metadata to version file data pointers and file attributes, and multiversion b-trees to version directory entries. We chose this division of methods based on the expected usage patterns of each. Assuming many versions of file attributes and a need to access them in their entirety most of the time, we decided that journal-based metadata would be more efficient. Directories, on the other hand, are updated less frequently than file metadata and a large fraction of operations are entry lookup rather than full listing. Thus, the cost of having history entries within the tree is expected to be lower.

Since the only pruning heuristic in CVFS is expiration, it requires a cleaner to find and remove expired versions. Although CVFS's background cleaner is not described in detail here, its implementation closely resembles the cleaner in LFS. The only added complication is that, when moving a data block in a versioning system, the cleaner must update all of the metadata versions that point to the block. Locating and modifying all of this metadata can be expensive. To address this problem, each data block on the disk is assigned a virtual block number. This allows us to move the physical location of the data and only have to update a single pointer within a virtual indirection table, rather than all of the associated metadata.

4.2 Layout and Allocation

Because of CVFS's log-structured format, disk space is managed in contiguous sets of disk blocks called *segments*. At any particular time, there is a single *write segment*. All data block allocations are done within this segment. Once the segment is completely allocated, a new write segment is chosen. Free segments on the disk are tracked using a bitmap.

As CVFS performs allocations from the write segment, the allocated blocks are marked as either journal blocks or data blocks. Journal blocks hold journal entries, and they contain pointers that string all of the journal blocks together into a single contiguous journal. Data blocks contain file data or metadata checkpoints.

CVFS uses inodes to store a file's metadata, including file size, access permissions, creation time, modification time, and the time of the oldest version still stored on the disk. The inode also holds direct and indirect data pointers for the associated file or directory. CVFS tracks inodes with a unique inode number. This inode number indexes into a table of inode pointers that are kept at a

Entry Type	Description	Cause
Attribute	Holds new inode attribute information	Inode change
Delete	Holds inode number and delete time	Inode change
Truncate	Holds the new size of the file	File data change
Write	Points to the new file data	File data change
Checkpoint	Points to checkpointed metadata	Metadata checkpoint / Inode change

Table 1: **Journal entry types.** This table lists the five types of journal entry. Journal entries are written when inodes are modified, file data is modified, or file metadata is flushed from the cache.

fixed location on the disk. Each pointer holds the block number of the most current metadata checkpoint for that file, which is guaranteed to hold the most current version of the file's inode. The in-memory copy of an inode is always kept up-to-date with the current version, allowing quick access for standard operations. To ensure that the current version can always be accessed directly off the disk, CVFS checkpoints the inode to disk on a cache flush.

4.3 The Journal

The string of journal blocks that runs through the segments of the disk is called the *journal*. Each journal block holds several time-ordered, variably-sized journal entries. CVFS uses the journal to implement both conventional file system journaling (a.k.a. write-ahead logging) and journal-based metadata.

Each journal entry contains information specific to a single change to a particular file. This information must be enough to do both roll-forward and roll-back of the metadata. Roll-forward is needed for update consistency in the face of failures. Roll-back is needed to reconstruct old versions. Each entry also contains the time at which the entry was written and a pointer to the location of the previous entry that applies to this particular file. This pointer allows us to trace the changes of a single file through time.

Table 1 lists the five different types of journal entries. CVFS writes entries in three different cases: inode modifications (creation, deletion, and attribute updates), data modifications (writing or truncating file data), and metadata checkpoints (due to a cache flush or history optimization).

4.4 Metadata

There are three types of file metadata that can be altered individually: inode attributes, file data pointers, and directory entries. Each has characteristics that match it to a particular method of metadata versioning.

4.4.1 Inode Attributes

There are four operations that act upon inode attributes: creation, deletion, attribute updates, and attribute lookups.

CVFS creates inodes by building an initial copy of the new inode and checkpointing it to the disk. Once this checkpoint completes and the inode pointer is updated, the file is accessible. The initial checkpoint entry is required because the inode cannot be read through the inode pointer table until a checkpoint occurs. CVFS's default checkpointing policy bounds the back-in-time access performance to approximately 150ms as is described in Section 5.3.2.

To delete an inode, CVFS writes a "delete" journal entry, which notes the inode number of the file being deleted. A flag is also set in the current version of the inode, specifying that the file was deleted, since the deleted inode cannot actually be removed from the disk until it expires.

CVFS stores attribute modifications entirely within a journal entry. This journal entry contains the value of the changed inode attributes both before and after the modification. Therefore, an attribute update involves writing a single journal entry, and updating the current version of the inode in memory.

CVFS accesses the current version of the attributes by reading in the current inode, since all of the attributes are stored within it. To access old versions of the attributes, CVFS traverses the journal entries searching for modifications that affect the attributes of that particular inode. Once roll-back is complete, the system is left with a copy of the attributes at the requested point in time.

4.4.2 File Data Pointers

CVFS tracks file data locations using direct and indirect pointers [30]. Each file's inode contains thirty direct pointers, as well as one single, one double and one triple indirect pointer.

When CVFS writes to a file, it allocates space for the new data within the current write segment and creates a "write" journal entry. The journal entry contains pointers to the data blocks within the segment, the range of logical block numbers that the data covers, the old size of the file, and pointers to the old data blocks that were overwritten (if there were any). Once the journal entry is allocated, CVFS updates the current version of the metadata to point at the new data.

If a write is larger than the amount of data that will fit within the current write segment, CVFS breaks the write into several data/journal entry pairs across different segments. This compartmentalization simplifies cleaning.

To truncate a file, CVFS first checkpoints the file to the log. This is necessary because CVFS must be able to locate truncated indirect blocks when reading back-in-time. If they are not checkpointed, then the information in them will be lost during the truncate; although earlier journal entries could be used to recreate this information, such entries could expire and leave the detection window, resulting in lost information. Once the checkpoint is complete, a "truncate" journal entry is created containing both a pointer to the checkpointed metadata and the new size of the file.

To access current file data, CVFS finds the most current inode and reads the data pointers directly, since they are guaranteed to be up-to-date. To access historical data versions, CVFS uses a combination of checkpoint tracking and journal roll-back to recreate the desired version of the requested data pointers. CVFS's checkpoint tracking and journal roll-back work together in the following way. Assume a user wishes to read data from a file at time T. First, CVFS locates the oldest checkpoint it is tracking with time T_c such that $T_c \geq T$. Next, it searches backward from that checkpoint through the journal looking for changes to the block numbers being read. If it finds an older version of a block that applies, it will use that. Otherwise, it reads the block from the checkpointed metadata.

To illustrate this journal rollback, Figure 4 shows a sequence of updates to block 3 of inode 4 interspersed with checkpoints of inode 4. Each block update and inode checkpoint is labeled with the time t that it was written. To read block 3 at time $T_1 = 12$, CVFS first reads the checkpoint at time $t = 18$, then reads journal entries to see if a different data block should be used. In this case, it finds that the block was overwritten at time $t = 15$, and so returns the older block written at time $t = 10$. In the case of time $T_2 = 5$, CVFS starts with the checkpoint at time $t = 7$, and then reads the journal entry, and realizes that no such block existed at time $t = 5$.

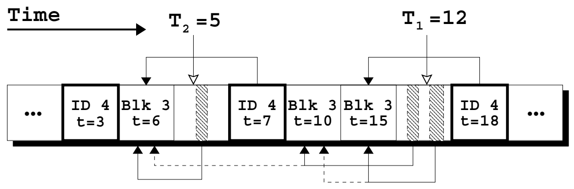


Figure 4: **Back-in-time access.** This diagram shows a series of checkpoints of inode 4 (highlighted with a dark border) and updates to block 3 of inode 4. Each checkpoint and update is marked with a time t at which the event occured. Each checkpoint holds a pointer to the block that is valid at the time of the checkpoint. Each update is accompanied by a journal entry (marked by thin, grey boxes) which holds a pointer to the new block (solid arrow) and the old block that it overwrote (dashed arrow, if one exists).

4.4.3 Directory Entries

Each directory in CVFS is implemented as a multiversion b-tree. Each entry in the tree represents a directory entry; therefore, each b-tree entry must contain the entry's name, the inode number of the associated file, and the time over which the entry is valid. Each entry also contains a fixed-size hash of the name. Although the actual name must be used as the key while searching through the entry blocks, this fixed-size hash allows the index blocks to use space-efficient fixed-size keys.

CVFS uses a full data block for each entry block of the tree, and sorts the entries within it first by hash and then by time. Index nodes of the tree are also full data blocks consisting of a set of index pointers also sorted by hash and then by time. Each index pointer is a ⟨subtree, hash, time-range⟩ tuple, where *subtree* is a pointer to the appropriate child block, *hash* is the name hash of the last entry along the subtree, and *time-range* is the time over which that same entry is valid.

With this structure, lookup and listing operations on the directory are the same as with a standard b-tree, except that the requested time of the operation becomes part of the key. For example, in Figure 3a, a lookup of $\langle C,\ 5\rangle$ searches through the tree for entries with name C, and then checks the time-ranges of each to determine the correct entry to return (in this case $\langle C,\ 4-7\rangle$). A listing of the directory at time 5 would do an in-order tree traversal (just as in a standard b-tree), but would exclude any entries that are not valid at time 5.

Insert, remove, and update are also very similar. Insert is identical, with the time-range of the new entry starting at the current time. Remove is an update of the time-range for the requested name. For example, in Figure 3b, entry E is removed at time 8. Update is an atomic remove and insert of the same entry name. For example, in Figure 3b, entry G is updated at time 9. This involves atomically removing the old entry G at time 9 (updating the time-range), and inserting entry G at time 9 (the new entry $\langle G,\ 9{-}?\rangle$).

Labyrinth Traces

		Versioned Data	Versioned Metadata	Metadata Savings	Total Savings
Files:	Conventional versioning	123.4 GB	142.4 GB		
	Journal-based metadata	123.4 GB	4.2 GB	97.1%	52.0%
Directories:	Conventional versioning	—	9.7 GB		
	Multiversion b-trees	—	0.044 GB	99.6%	99.6%
Total:	Conventional versioning	123.4 GB	152.1 GB		
	CVFS	123.4 GB	4.244 GB	**97.2%**	**53.7%**

Lair Traces

		Versioned Data	Versioned Metadata	Metadata Savings	Total Savings
Files:	Conventional versioning	74.5 GB	34.8 GB		
	Journal-based metadata	74.5 GB	1.1 GB	96.8%	30.8%
Directories:	Conventional versioning	—	1.8 GB		
	Multiversion b-trees	—	0.0064 GB	99.7%	99.7%
Total:	Conventional versioning	74.5 GB	36.6 GB		
	CVFS	74.5 GB	1.1064 GB	**97.0%**	**32.0%**

Table 2: **Space utilization.** This table compares the space utilization of conventional versioning with CVFS, which uses journal-based metadata and multiversion b-trees. The space utilization for versioned data is identical for conventional versioning and journal-based metadata because neither address data beyond block sharing. Directories contain no versioned data because they are entirely a metadata construct.

5 Evaluation

The objective of this work is to reduce the space overheads of versioning without reducing the performance of current version access. Therefore, our evaluation of CVFS is done in two parts. First, we analyze the space utilization of CVFS. We find that using journal-based metadata and multiversion b-trees reduces space overhead for versioned metadata by over 80%. Second, we analyze the performance characteristics of CVFS. We find that it performs similarly to non-versioning systems for current version access, and that back-in-time performance can be bounded to acceptable levels.

5.1 Experimental Setup

For the evaluation, we used CVFS as the underlying file system for S4, our self-securing NFS server. S4 is a user-level NFS server written for Linux that uses the SCSI-generic interface to directly access the disk. S4 exports an NFSv2 interface and treats it as a security perimeter between the storage system and the client operating system. Although the NFSv2 specification requires that all changes be synchronous, S4 also has an asynchronous mode of operation, allowing us to more thoroughly analyze the performance overheads of our metadata versioning techniques.

In all experiments, the client system has a 550 MHz Pentium III, 128 MB RAM, and a 3Com 3C905B 100 Mb network adapter. The servers have two 700 MHz Pentium IIIs, 512 MB RAM, a 9 GB 10,000 RPM Quantum Atlas 10K II drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro100 100 Mb network adapter. The client and server are on the same 100 Mb network switch.

5.2 Space Utilization

We used two traces, labelled *Labyrinth* and *Lair*, to evaluate the space utilization of our system. The *Labyrinth* trace is from an NFS server at Carnegie Mellon holding the home directories and CVS repository that support the activities of approximately 30 graduate students and faculty; it records approximately 164 GB of data traffic to the NFS server over a one-month period. The *Lair* trace [13] is from a similar environment at Harvard; it records approximately 103 GB of data traffic over a one-week period. Both were captured via passive network monitoring.

We replayed each trace onto both a standard configuration of CVFS and a modified version of CVFS. The modified version simulates a conventional versioning system by checkpointing the metadata with each modification. It also performs copy-on-write of directory blocks, overwriting the entries in the new blocks (that is, it uses normal b-trees). By observing the amount of allocated data for each request, we calculated the exact overheads of our two metadata versioning schemes as compared to a conventional system.

Table 2 compares the space utilization of versioned files for the two traces using conventional versioning and journal-based metadata. There are two space overheads for file versioning: versioned data and versioned metadata. The overhead of versioned data is the overwritten or deleted data blocks that are retained. In both

Labyrinth Traces							
		Versioned Data	**Versioned File Metadata**	**Versioned Directories**	**Version Ratio**	**Metadata Savings**	**Total Savings**
Comprehensive:	Conventional	123.4 GB	142.4 GB	9.7 GB			
	CVFS	123.4 GB	4.2 GB	0.044 GB	1:1	97%	54%
On close():	Conventional	55.3 GB	30.6 GB	2.4 GB			
	CVFS	55.3 GB	2.1 GB	0.012 GB	1:2.8	94%	35%
6 minute	Conventional	53.2 GB	11.0 GB	2.4 GB			
Snapshots:	CVFS	53.2 GB	1.3 GB	0.012 GB	1:11.7	90%	18%
1 hour	Conventional	49.7 GB	5.1 GB	2.4 GB			
Snapshots:	CVFS	49.7 GB	0.74 GB	0.012 GB	1:20.8	90%	12%

Lair Traces							
		Versioned Data	**Versioned File Metadata**	**Versioned Directories**	**Version Ratio**	**Metadata Savings**	**Total Savings**
Comprehensive:	Conventional	74.5 GB	34.8 GB	1.79 GB			
	CVFS	74.5 GB	1.1 GB	0.0064 GB	1:1	97%	32%
On close():	Conventional	40.3 GB	6.1 GB	0.75 GB			
	CVFS	40.3 GB	0.57 GB	0.0032 GB	1:2.9	91%	13%
6 minute	Conventional	38.2 GB	3.0 GB	0.75 GB			
Snapshots:	CVFS	38.2 GB	0.36 GB	0.0032 GB	1:11.2	88%	8%
1 hour	Conventional	36.2 GB	2.0 GB	0.75 GB			
Snapshots:	CVFS	36.2 GB	0.26 GB	0.0032 GB	1:15.6	87%	6%

Table 3: **Benefits for different versioning schemes.** This table shows the benefits of journal-based metadata for three versioning schemes that use pruning heuristics. For each scheme it compares conventional versioning with CVFS's journal-based metadata and multiversion b-trees, showing the versioned metadata sizes, the corresponding metadata savings, and the total space savings. It also displays the ratio of versions to file modifications; more modifications per version generally reduces both the importance and the compressibility of versioned metadata.

conventional versioning and journal-based metadata, the versioned data consumes the same amount of space, since both schemes use block sharing for versioned data. The overhead of versioned metadata is the information needed to track the versioned data. For *Labyrinth*, the versioned metadata consumes as much space as the versioned data. For *Lair*, it consumes only half as much space as the versioned data, because *Lair* uses a larger block size; on average, twice as much data is overwritten with each WRITE.

Journal-based metadata: Journal-based metadata reduces the space required for versioned file metadata substantially. For the conventional system, versioned metadata consists of copied inodes and sometimes indirect blocks. For journal-based metadata, it is the log entries that allow recreation of old versions plus any checkpoints used to improve back-in-time performance (see Section 5.3.2). For both traces, this results in 97% reduction of space required for versioned metadata.

Multiversion b-trees: Using multiversion b-trees for directories provides even larger space utilization reductions. Because directories are a metadata construct, there is no versioned data. The overhead of versioned metadata in directories is the space used to store the overwritten and deleted directory entries. In a conventional versioning system, each entry creation, modification, or removal results in a new block being written that contains the change. Since the entire block must be kept over the detection window, it results in approximately 9.7 GB of space for versioned entries in the *Labyrinth* trace and 1.8 GB in the *Lair* trace. With multiversion b-trees, the only overhead is keeping the extra entries in the tree, which results in approximately 45 MB and 7 MB of space for versioned entries in the respective traces.

5.2.1 Other Versioning Schemes

We also use the *Labyrinth* and *Lair* traces to compute the space that would be required to track versions in three other versioning schemes: versioning on every file CLOSE, taking systems snapshots every 6 minutes, and taking system snapshots every hour. In order to simulate open-close semantics with our NFS server, we insert a CLOSE call after sequences of operations on a given file that are followed by 500ms of inactivity to that file.

Table 3 shows the benefits of CVFS's mechanisms for the three versioning schemes mentioned above. For each scheme, the table also shows the ratio of file versions-to-modifications (e.g., in comprehensive versioning, each modification results in a new version, so the ratio is 1:1). For on-close versioning in the *Labyrinth* trace, conventional versioning requires 55% as much space for versioned metadata as versioned data, meaning that reduction can still provide large benefits. As the versioned metadata to versioned data ratio decreases and as the version ratio increases, the overall benefits of versioned metadata compression drop.

Table 3 identifies the benefits of both journal-based metadata (for "Versioned File Metadata") and multiversion b-trees (for "Versioned Directories"). For both, the metadata compression ratios are similar to those for comprehensive versioning. The journal-based metadata ratio drops slightly as the version ratio increases, because capturing more changes to the file metadata moves the journal entry size closer to the actual metadata size. The multiversion b-tree ratio is lower because a most of the directory updates fall into one of two categories: entries that are permanently added or temporary entries that are created and then rapidly renamed or deleted. For this reason, the number of versioned entries is lower for other versioning schemes; although multiversion b-trees use less space, the effect on overall savings is reduced.

5.3 Performance Overheads

The performance evaluation is done in three parts. First, we compare the S4 prototype to non-versioning systems using several macro benchmarks. Second, we measure the back-in-time performance characteristics of journal-based metadata. Third, we measure the general performance characteristics of multiversion b-trees.

5.3.1 General Comparison

The purpose of the general comparison is to verify that the S4 prototype performs comparably to non-versioning systems. Since part of our objective is to avoid undue performance overheads for versioning, it is important that we confirm that the prototype performs reasonably relative to similar systems. To evaluate the performance relationship between S4 and non-versioning systems, we ran two macro benchmarks designed to simulate realistic workloads.

For both, we compare S4 in both synchronous and asynchronous modes against three other systems: a NetBSD NFS server running FFS, a NetBSD NFS server running LFS, and a Linux NFS server running EXT2. We chose to compare against BSD's LFS because it uses a log-structured layout similar to S4's. BSD's FFS and Linux's EXT2 use similar, more "traditional" file layout techniques that differ from S4's log-structured layout. It is not our intent to compare a LFS layout against other layouts, but rather to confirm that our implementation does not have any significant performance anomalies. To ensure this, a small discussion of the performance differences between the systems is given for each benchmark.

Each of these systems was measured using an NFS client running on Linux. Our S4 measurements use the S4 server and a Linux client. For "Linux," we run RedHat 6.1 with a 2.2.17 kernel. For "NetBSD," we run a stock NetBSD 1.5 installation.

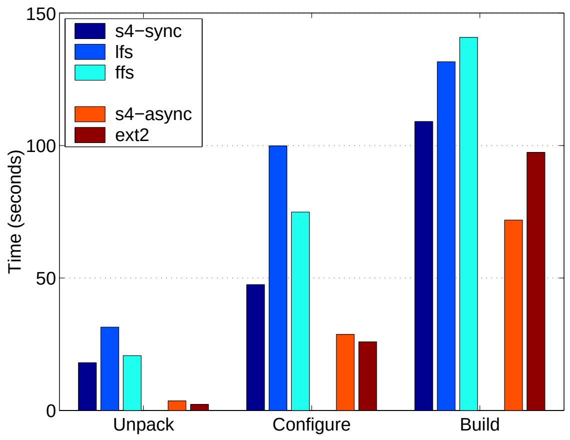


Figure 5: **SSH comparison.** This figure shows the performance of five systems on the unpack, configure, and build phases of the SSH-build benchmark. Performance is measured in the elapsed time of the phase. Each result is the average of 15 runs, and all variances are under .5s with the exception of the build phases of ffs and lfs which had variances of 37.6s and 65.8s respectively.

To focus comparisons, the five setups should be viewed in two groups. BSD LFS, BSD FFS, and S4-sync all push updates to disk synchronously, as required by the NFSv2 specification. Linux EXT2 and S4-async do not; instead, updates are made in memory and propagated to disk in the background.

SSH-build [39] was constructed as a replacement for the Andrew file system benchmark [20]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v1.2.27 (approximately 1 MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Both the server and client caches are flushed between phases.

Figure 5 shows the SSH-build results for each of the five different systems. As we hoped, our S4 prototype performs similarly to the other systems measured.

LFS does significantly worse on unpack and configure because it has poor small write performance. This is due to the fact that NetBSD's LFS implementation uses a 1 MB segment size, and NetBSD's NFS server requires a full sync of this segment with each modification; S4 uses a 64kB segment size, and supports partial segments. Adding these features to NetBSD's LFS implementation would result in performance similar to S4[1]. FFS performs worse than S4 because FFS must update both a data block and inode with each file modification, which are in separate locations on the disk. EXT2 performs more closely to S4 in asynchronous mode because it fails to satisfy NFS's requirement of synchronous modifications. It does slightly better in the unpack and configure stages because it maintains no consistency guarantees, however it does worse in the build phase due to S4's segment-sized reads.

Postmark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [22]. It creates a large number of small randomly-sized files (between 512 B and 9 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 20,000 transactions on 5,000 files, and the biases for transaction types are equal.

Figure 6 shows the Postmark results for the five server configurations. These show similar results to the SSH-build benchmark. Again, S4 performs comparably. In particular, LFS continues to perform poorly due to its small write performance penalty caused by its interaction with NFS. FFS still pays its performance penalty due to multiple updates per file create or delete. EXT2 performs even better in this benchmark because the random, small file accesses done in Postmark are not assisted by aggressive prefetching, unlike the sequential, larger accesses done during a compilation; however, S4 continues to pay the cost of doing larger accesses, while EXT2 does not.

5.3.2 Journal-based Metadata

Because the metadata structure of a file's current version is the same in both journal-based metadata and conventional versioning systems, their current version access times are identical. Given this, our performance measurements focus on the performance of back-in-time operations with journal-based metadata.

There are two main factors that affect the performance of back-in-time operations: checkpointing and clustering. Checkpointing refers to the frequency of metadata checkpoints. Since journal roll-back can begin with any checkpoint, CVFS keeps a list of metadata checkpoints for each file, allowing it to start roll-back from the closest checkpoint. The more frequently CVFS creates checkpoints, the better the back-in-time performance.

Clustering refers to the physical distance between relevant journal entries. With CVFS's log-structured layout, if several changes are made to a file in a short span of time, then the journal entries for these changes are likely to be clustered together in a single segment. If several journal entries are clustered in a single segment together, then they are all read together, speeding up journal roll-back. The "higher" the clustering, the better the performance is expected to be.

[1]We tried changing the NetBSD LFS segment size, but it was not stable enough to complete any benchmark runs.

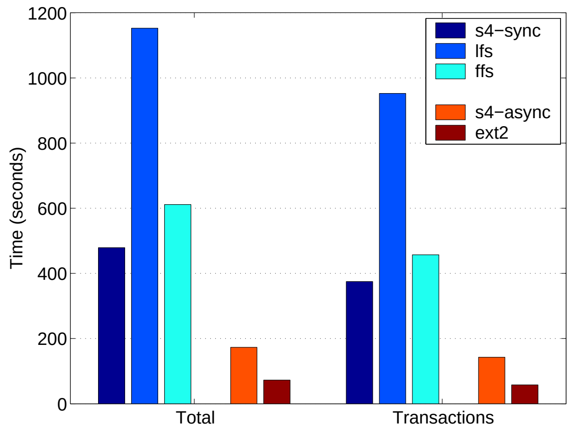


Figure 6: **Postmark comparison.** This figure shows the the elapsed time for both the entire run of postmark and the transactions phase of postmark for the five test systems. Each result is the average of 15 runs, and all variances are under 1.5s

Figure 7 shows the back-in-time performance characteristics of journal-based metadata. This graph shows the access time in milliseconds for a particular version number of a file back-in-time. For example, in the worst-case, accessing the 60th version back-in-time would take 350ms. The graph examines four different scenarios: best-case behavior, worst-case behavior, and two potential cases (one involving low clustering and one involving high clustering).

The best-case back-in-time performance is the situation where a checkpoint is kept for each version of the file, and so any version can be immediately accessed with no journal roll-back. This is the performance of a conventional versioning system. The worst-case performance is the situation where no checkpoints are kept, and every version must be created through journal roll-back. In addition there is no clustering, since each journal entry is in a separate segment on the disk. This results in a separate disk access to read each entry. In the high clustering case, changes are made in bursts, causing journal entries to be clustered together into segments. This reduces the slope of the back-in-time performance curve. In the low clustering case, journal entries are spread more evenly across the segments, giving a higher slope. In both the low and high clustering cases, the points where the performance drops back to the best-case are the locations of

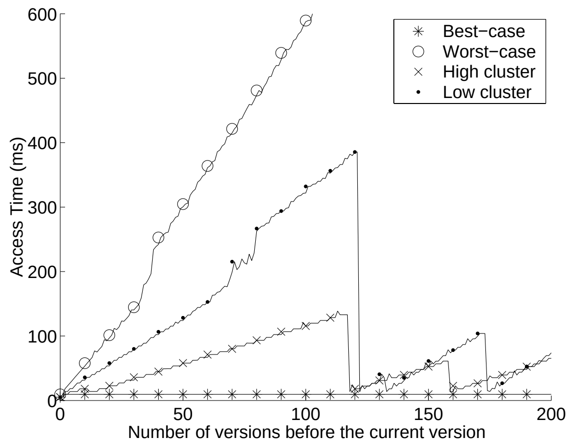


Figure 7: **Journal-based metadata back-in-time performance.** This figure shows several potential curves for back-in-time performance of accessing a single 1KB file. The worst-case is when journal roll-back is used exclusively, and each journal entry is in a separate segment on the disk. The best-case is if a checkpoint is available for each version, as in a conventional versioning system. The high and low clustering cases are examples of how checkpointing and access patterns can affect back-in-time performance. Both of these cases use random checkpointing. In the “high cluster” case, there are an average of 5 versions in a segment. In the “low cluster” case, there are an average of 2 versions in a segment. The cliffs in these curves indicate the locations of checkpoints, since the access time for a checkpointed version drops to the best-case performance. As the level of clustering increases, the slope of the curve decreases, since multiple journal entries are read together in a single segment. Each curve is the average of 5 runs, and all variances are under 1ms.

checkpoints.

Using this knowledge of back-in-time performance, a system can perform a few optimizations. By tracking checkpoint frequency and journal entry clustering, CVFS can predict the back-in-time performance of a file while it is being written. With this information, CVFS bounds the performance of the back-in-time operations for a particular file by forcing a checkpoint whenever back-in-time performance is expected to be poor. For example, in Figure 7, the high-clustering case keeps checkpoints in such a way as to bound back-in-time performance to around 100ms at worst. In our S4 prototype, we bound the back-in-time performance to approximately 150ms. Another possibility is to keep checkpoints at the point at which one believes the user would wish to access the file. Using a heuristic such as in the Elephant FS [37] to decide when to create file checkpoints might closely simulate the back-in-time performance of conventional versioning.

5.3.3 Multiversion B-trees

Figure 8 shows the average access time of a single entry from a directory given some fixed number of entries currently stored within the directory (notice the log scale of the x-axis). To see how a multiversion b-tree performs as compared to a standard b-tree, we must compare two different points on the graph. The point on the graph corresponding to the number of current entries in the directory represents the access time of a standard b-tree. The point on the graph corresponding to the combined number of current and history entries represents the access time of a multiversion b-tree. The difference between these values is the lookup performance lost by keeping the extra versions.

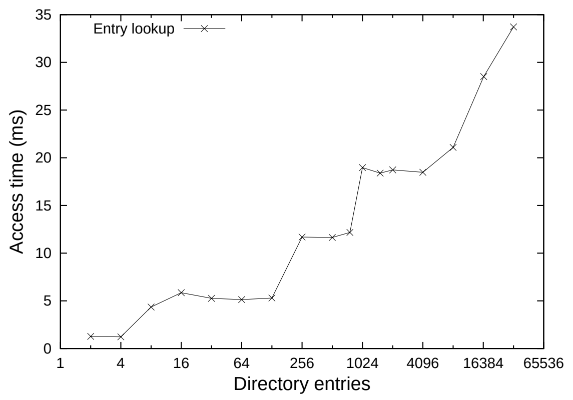


Figure 8: **Directory entry performance.** This figure shows the average time to access a single entry out of the directory given the total number of entries within the directory. History entries affect performance by increasing the effective number of entries within the directory. The larger the ratio of history entries to current entries, the more current version performance will suffer. This curve is the average of 15 runs and the variance for each point is under .2ms.

Using the traces gathered from our NFS server, we found that the average number of current entries in a directory is approximately 16. Given a detection window of one month, the number of history entries is less than 100 over 99% of the time, and between zero and five over 95% of the time. Since approximately 200 entries can fit into a block, there is generally no performance lost by keeping the history. This block-based performance explains the stepped nature of Figure 8.

5.4 Summary

Our results show that CVFS reduces the space utilization of versioned metadata by more than 80% without causing noticeable performance degradation to current version access. In addition, through intelligent checkpointing, it is possible to bound back-in-time performance to within a constant factor of conventional versioning systems.

6 Metadata Versioning in Non-Log-Structured Systems

Most file systems today use a layout scheme similar to that of BSD FFS rather than a log-structured layout. Such systems can be extended to support versioning relatively easily; Santry et al. [37] describe how this was done for Elephant, including tracking of versions and block sharing among versions. For non-trivial pruning policies, such as those used in Elephant and CVFS, a cleaner is required to expire old file versions. In an FFS-like system, unlike in LFS, the cleaner does not necessarily have the additional task of coalescing free space.

Both journal-based metadata and multiversion b-trees can be used in a versioning FFS-like file system in much the same way as in CVFS. Replacing conventional directories with multiversion b-trees is a self-contained change, and the characteristics should be as described in the paper. Replacing replicated metadata versions with journal-based metadata requires effort similar to adding write-ahead logging support. Experience with adding such support [17, 39] suggests that relatively little effort is involved, little change to the on-disk structures is involved, and substantial benefits accrue. If such logging is already present, journal-based metadata can piggyback on it.

Given write-ahead logging support, journal-based metadata requires three things. First, updates to the write-ahead log, which are the journal entries, must contain enough information to roll-back as well as roll-forward. Second, the journal entries must be kept until the cleaner removes them; they cannot be removed via standard log checkpointing. This will increase the amount of space required for the log, but by much less than the space required to instead retain metadata replicas. Third, the metadata replica support must be retained for use in tracking metadata version checkpoints.

With a clean slate, we think an LFS-based design is superior if many versions are to be retained. Each version committed to disk requires multiple updates, and LFS coalesces those updates into a single disk write. LFS does come with cleaning and fragmentation issues, but researchers have developed sufficiently reasonable solutions to them to make the benefits outweigh the costs in many environments. FFS-type systems that employ copy-on-write versioning have similar fragmentation issues.

7 Related Work

Much work has been done in the areas of versioning and versioned data structures, log-structured file systems, and journaling.

Several file systems have used versioning to provide recovery from both user errors and system failure. Both Cedar [17] and VMS [29] use file systems that offer simple versioning heuristics to help users recover from their mistakes. The more recent Elephant file system provides a more complete range of versioning options for recovery from user error [37]. Its heuristics attempt to keep only those versions of a file that are most important to users.

Many modern systems support snapshots to assist recovery from system failure [11, 19, 20, 25, 34]. Most closely related to CVFS are Spiralog [15, 21] and Plan9 [33], which use a log-structured file system to do online backup by recording the entire log to tertiary storage. Chervenak, et al., performed an evaluation of several snapshot systems [10].

Version control systems are user programs that implement a versioning system on top of a traditional file system [16, 27, 43]. These systems store the current version of the file, along with differences that can be applied to retrieve old versions. These systems usually have no concept of checkpointing, and so recreating old versions is expensive.

Write-once storage media keeps a copy of any data written to it. The Plan 9 system [33] utilized this media to permanently retain all filesystem snapshots using a log-structured technique. A recent improvement to this method is the Venti archival storage system. Venti creates a hash of each block written and uses that as a unique identifier to map identical data blocks onto the same physical location [34]. This removes the need to rewrite identical blocks, reducing the space required by individual data versions and files that contain similar data. It is interesting to consider combining Venti's data versioning with CVFS's metadata structures to provide extremely space efficient comprehensive versioning.

In addition to the significant file system work in versioning, there has been quite a bit of work done in the database community for keeping versions of data through time. Most of this work has been done in the form of "temporal" data structures [2, 23, 24, 44, 45]. Our directory structure borrows from these techniques.

The log-structured data layout was developed for write-once media [33], and later extended to provide write performance benefits for read-write disk technology [36]. Since its inception, LFS has been evaluated [3, 28, 35, 38] and used [1, 7, 12, 18] by many different groups. Much of the work done to improve both LFS and LFS cleaners is directly applicable to CVFS.

While journal-based metadata is a new concept, journaling has been used in several different file systems to provide metadata consistency guarantees efficiently [8, 9, 11, 39, 42]. Similarly to journal-based metadata, LFS's segment summary block contains all of the metadata for the data in a segment, but is stored in an uncompressed format. Zebra's deltas improved upon this by storing only the changes to the metadata, but were designed exclusively for roll-forward (a write-ahead log). Database systems also use the roll-back and roll-forward concepts to ensure consistency during transactions with commit and abort [14].

Several systems have used copy-on-write and differencing techniques that are common to versioning systems to decrease the bandwidth required during system backup or distributed version updates [4, 6, 26, 31, 32]. Some of these data differencing techniques [5, 26, 31] could be applied to CVFS to reduce the space utilization of versioned data.

8 Conclusion

This paper shows that journal-based metadata and multiversion b-trees address the space-inefficiency of conventional versioning. Integrating them into the CVFS file system has nearly doubled the detection window that can be provided with a given storage capacity. Further, current version performance is affected minimally, and back-in-time performance can be bounded reasonably with checkpointing.

Acknowledgments

We would like to thank our shepard Mike Franklin and the anonymous reviewers for their comments and helpful suggestions. Also, Dan Ellard and others at Harvard for providing the *Lair* trace used in this work. We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. Garth Goodson has been supported by an IBM Fellowship. Craig Soules has been supported by a Usenix Association Scholarship. This material is based in part on research sponsored by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by the DARPA/ITO OASIS program (Air Force contract number F30602-99-2-0539-AFRL).[2]

[2]The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5):109–126, 1995.

[2] B. Becker, S. Gschwind, T. Ohler, P. Widmayer, and B. Seeger. An asymptotically optimal multiversion b-tree. *Very large data bases journal*, **5**(4):264–275, 1996.

[3] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. USENIX Annual Technical Conference, pages 277–288. USENIX Association, 1995.

[4] R. C. Burns. Version management and recoverability for large object data. International Workshop on Multimedia Database Management, pages 12–19. IEEE Computer Society, 1998.

[5] R. C. Burns. *Differential compression: a generalized solution for binary files*. Masters thesis. University of California at Santa Cruz, December 1996.

[6] R. C. Burns and D. D. E. Long. Efficient distributed backup with delta compression. Workshop on Input/Output in Parallel and Distributed Systems, pages 26–36. ACM Press, December 1997.

[7] M. Burrows, C. Jerian, B. Lampson, and T. Mann. *Online data compression in a log-structured file system*. 85. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, April 1992.

[8] L. F. Cabrera, B. Andrew, K. Peltonen, and N. Kusters. Advances in Windows NT storage management. *Computer*, **31**(10):48–54, October 1998.

[9] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors. *IBM Journal of Research and Development*, **34**(1):105–110, January 1990.

[10] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: a survey of backup techniques. Joint NASA and IEEE Mass Storage Conference, 1998.

[11] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. USENIX Annual Technical Conference, pages 43–60, 1992.

[12] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: a new approach to improving file systems. ACM Symposium on Operating System Principles, pages 15–28, 1993.

[13] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of an Email and Research Workload. Conference on File and Storage Technologies. USENIX Association, 2003.

[14] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Potzulo, and I. Traiger. The recovery manager of the System R database manager. *ACM Computing*

Surveys, **13**(2):223–242, June 1981.

[15] R. J. Green, A. C. Baird, and J. Christopher. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal*, **8**(2):32–45, 1996.

[16] D. Grune, B. Berliner, and J. Polk. Concurrent Versioning System, http://www.cvshome.org/.

[17] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **21**(5):155–162, 1987.

[18] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, **13**(3):274–310. ACM Press, August 1995.

[19] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. Winter USENIX Technical Conference, pages 235–246. USENIX Association, 1994.

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

[21] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, **8**(2):5–14, 1996.

[22] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[23] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering*, **10**(1), February 1998.

[24] S. Lanka and E. Mays. Fully Persistent B+-trees. ACM SIGMOD International Conference on Management of Data, pages 426–435. ACM, 1991.

[25] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[26] J. MacDonald. *File system support for delta compression*. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[27] J. MacDonald, P. N. Hilfinger, and L. Semenzato. PRCS: The project revision control system. European Conference on Object-Oriented Programming. Published as *Proceedings of ECOOP*, pages 33–45. Springer-Verlag, 1998.

[28] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **31**(5):238–252. ACM, 1997.

[29] K. McCoy. *VMS file system internals*. Digital Press, 1990.

[30] 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.

[31] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. ACM Symposium on Operating System Principles. Published as *Operating System Review*, **35**(5):174–187. ACM, 2001.

[32] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. Conference on File and Storage Technologies, pages 117–129. USENIX Association, 2002.

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

[34] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. Conference on File and Storage Technologies, pages 89–101. USENIX Association, 2002.

[35] J. T. Robinson. Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning. *Operating Systems Review*, **30**(4):29–32, October 1996.

[36] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **25**(5):1–15, 1991.

[37] D. S. Santry, M. J. Feeley, N. C. Hutchinson, R. W. Carton, J. Ofir, and A. C. Veitch. Deciding when to forget in the Elephant file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **33**(5):110–123. ACM, 1999.

[38] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: a performance comparison. USENIX Annual Technical Conference, pages 249–264. Usenix Association, 1995.

[39] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. USENIX Annual Technical Conference, 2000.

[40] J. D. Strunk, G. R. Goodson, A. G. Pennington, C. A. N. Soules, and G. R. Ganger. *Intrusion detection, diagnosis, and recovery with self-securing storage*. Technical report CMU–CS–02–140. Carnegie Mellon University, 2002.

[41] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation, pages 165–180. USENIX Association, 2000.

[42] A. Sweeney. Scalability in the XFS file system. USENIX Annual Technical Conference, pages 1–14, 1996.

[43] W. F. Tichy. *Software development control based on system structure description*. PhD thesis. Carnegie-Mellon University, Pittsburgh, PA, January 1980.

[44] V. J. Tsotras and N. Kangelaris. The snapshot index - an I/O-optimal access method for timeslice queries. *Information Systems*, **20**(3):237–260, May 1995.

[45] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, **9**(3). IEEE, May 1997.