Ext3cow: The Design, Implementation, and Analysis of Metadata for a Time-Shifting File System

Zachary N. J. Peterson　　Randal C. Burns

{*zachary,randal*}*@cs.jhu.edu*

Technical Report HSSL-2003-03
Hopkins Storage Systems Lab
Department of Computer Science
The Johns Hopkins University

Abstract

The ext3cow file system, built on Linux's popular ext3 file system, brings snapshot functionality and file versioning to the open-source community. Our implementation of ext3cow has several desirable properties: ext3cow is implemented entirely in the file system and, therefore, does not modify kernel interfaces or change the operation of other file systems; ext3cow provides a *time-shifting* interface that permits access to data in the past without polluting the file system namespace; and, ext3cow creates versions of files on disk without copying data in memory. Experimental results show that the time-shifting functions of ext3cow do not degrade file system performance. Ext3cow performs comparably to ext3 on many file system benchmarks and trace driven experiments.

1 Introduction

Storage and file systems use data versioning to enhance reliability, availability, and operational semantics. Versioning techniques include volume and file system snapshot as well as per file versioning. A snapshot is a read-only, immutable, and logical image of a collection of data as it appeared at a single point in time. Point-in-time snapshots of a file system are useful for consistent image for backup [6, 12, 15, 18] and for archiving and data mining [33]. File versioning, creating new logical versions on every disk write or on every open/close session, is used for tamper-resistant storage [43, 44] and file-oriented recovery from deletion [39]. Both techniques speed recovery and limit exposure to data losses during file system failure [17, 41], A range of snapshot implementations exist, both at the logical file system level [12, 18, 19, 33, 39] and the disk storage level [8, 16, 19, 44].

Existing systems have several shortcomings. Many software implementations are not modular, because they replace the virtual file system (VFS) switching layer with their own versioning interface. Changes to the VFS affect all file systems sharing a platform and degrade multiple file system support. A VFS is virtual and, therefore, does not manage storage directly. The only way a versioning system operating within the VFS layer can create versions is by creating new files in underlying file systems. Therefore, these designs pollute the namespace with named versions. Other snapshot implementations have awkward user and administration interfaces that place snapshots in their own logical volumes or require snapshots to be mounted as separate file systems. Lastly, some systems require specialized, and often expensive hardware, making these systems unattractive for the consumer.

This paper describes the design and implementation of *ext3cow*, an open-source disk file system based on the third extended file system (ext3). Ext3 [4] is the mature and stable file system originally designed for the Linux operating system, and is the default file system on most Linux distributions. Originally based on the Minix file system [45] and influenced by the Fast File System (FFS) [23], ext3 has become robust and reliable, providing reasonable performance and scalability for many users and workloads [3].

Ext3cow extends the ext3 design by retrofitting the in-memory and on-disk metadata structures to support versioning and by providing a user interface to versioning that is transparent to the operating system. Ext3cow implements snapshot and file versioning through fine-grained copy-on-write of file system blocks and inodes on disk. Ext3cow does not copy data in the buffer or page caches and, therefore, does not degrade cache performance. File versions and snapshots are accessed through a *time-shifting* interface that allows users to specify a file or directory at any

```
[user@machine] echo "This is the original foo.txt" > foo.txt
[user@machine] snapshot
Snapshot on .  1057845484
[user@machine] echo "This is the new foo.txt." > foo.txt
[user@machine] cat foo@1057845484
This is the original foo.txt.
[user@machine] cat foo
This is the new foo.txt.
```

Figure 1: Creating snapshots and accessing data in the past in ext3cow.

```
[user@machine] snapshot /usr/bin
Snapshot on /usr/bin 1057866367
[user@machine] ln -s /usr/bin@1057866367 /usr/bin.preinstall
[user@machine] /usr/bin.preinstall/gcc
```

Figure 2: Creating distinguished (named) snapshots in ext3cow.

point-in-time.

Our implementation of ext3cow serves as proof that versioning can be encapsulated entirely within an on-disk file system. This was our original goal and is the best feature of ext3cow. Encapsulation ensures that all kernel interfaces to the VFS and page cache remain the same and, by implication, that ext3cow co-exists with other Linux file systems.

We show that the retrofitted data structures and versioning policies have a minor effect on the file system performance. On most microbenchmarks, ext3cow meets or exceeds the performance of ext3. Using a traced workload over various snapshot frequencies, we observe a 4.1% to 4.7% increase in the number of inodes needed to support snapshot as compared to ext3. This result is consistent with the literature [43].

2 Time-shifting

Our goals in creating an interface to data versioning include offering rich semantics, making it congruent with operating system kernel interfaces, and providing access to all versions from within the file system. Semantically rich means that the way in which data are accessed provides insight into the age of the data. In the time-shifting principle, date and time information are embedded into the access path. The interface allows a user to fetch any file or directory from any point in time and to navigate the file system in the past.

Previous interfaces fail to fulfill our requirements for versioning. Some require old data to be accessed through a separate mount point [8, 43, 44], which prevents browsing in the existing file namespace to locate objects and then shifting those objects into the past. Others use arbitrary version numbers to access old data [7,12,15,21,26], *e.g.* access the file four versions back. These interfaces make sense for daily snapshots, but do not generalize to file versioning or more frequent snapshots. Others use namespace tunnels from the present to the past, such as a `.snapshot` directory [17] that accesses the snapshot version of the current directory. While this permits browsing for files in the present and then shifting those files to the past, it does not handle multiple versions gracefully.

We describe the operation of the time shifting interface through the example of Figure 1. A call to the `snapshot` utility causes a snapshot of the file system to be taken and returns the *snapshot epoch* `1057845484`. For epoch numbers, we use the number of seconds since the Epoch (00:00:00 UTC, January 1, 1970), which may be acquired through `gettimeofday`. Subsequent writes to the file cause the current version to be updated, but the version of the file at the snapshot is unchanged. To access the snapshot version, a user or application appends the @ symbol to the name and specifies a time. `Snapshot` is a user space program and library call that invokes a file-system specific `ioctl`, instructing ext3cow to create a snapshot. Using `ioctl` allows `snapshot` to bypass the virtual file system and communicate with ext3cow directly, which is consistent with our ethic of making no changes to kernel structures.

The time-shifting interface meets our requirements. Users and applications specify a day, hour, and second at which they want a file. The interface does not require the specified time to be exactly on a snapshot. Rather, the interface treats time continuously. Requesting a file at a time returns the file contents at the preceding snapshot. The interface uses the @ symbol, a legal symbol for file names, so that the VFS accepts the name and passes it through to ext3cow unmodified. The interface adds no new names to the namespace.

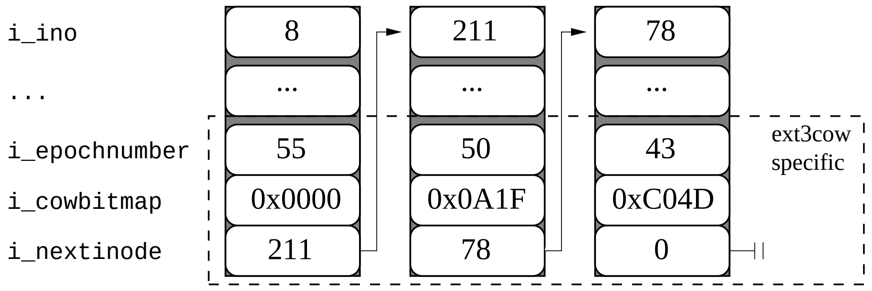


Figure 3: Both on-disk and in-memory inodes were retrofitted to support snapshot and copy-on-write by adding three fields: an inode epoch number, a copy-on-write bitmap, and field pointing to the next inode in the version chain.

Distinguished snapshots may be created using links, which allows time-shifting to emulate the behavior of systems that put snapshots in their own namespaces or mount snapshot namespaces in separate volumes. For example, an administrator might create a read-only version of a file system prior to installing new software (Figure 2). If installing software breaks `gcc`, the administrator can use the old `gcc` through the mounted snapshot. Because `@` is a legal file system symbol, the link can be placed anywhere in the namespace, even within another file system. Hard links may also be used to connect a name directly to an old inode. This example illustrates that time-shifting is inherited from the parent directory, *i.e.* the entire subtree below `/usr/bin.preinstall` is scoped to the snapshot. As an aside, it is permissible to have multiple time-shifts in a single path, *e.g.*
`/A/B@time1/C/D@time2/E`.

The time-shifting interface imposes some restrictions. The use of `gettimeofday` limits (named) snapshots to one per second. For systems that use snapshot as part of recovery [17], sub-second granularity may be necessary. Because ext3cow, like ext3, uses a journal for file system recovery, we found no emergent need for sub-second snapshot. The use of `@` has backward compatibility issues. Because ext3cow interprets `@` as a time specifier, applications are not be able to create files with `@` in the name.[1] Finally, while `gettimeofday` is a natural programmatic interface, users may find it unintuitive and prefer to work with other date/time formats. To remedy this, our future plans include time-shifting shell extensions that will support a variety of date, time and naming formats to help users browse versions.

This approach of accessing files by time is similar to the Elephant file system [38,39]. However, Elephant complicates the interface by requiring users to specify which versions of a file are important to them. Further, the Elephant interface does not provide for a continuous view of time.

3 Metadata Design

The metadata design of ext3cow supports the continuous time notion of the time-shifting interface within the framework of the data structures of the Linux VFS. Unlike many other snapshot file systems, ext3cow does not interfere or replace the Linux common file model, therefore, it integrates easily, requiring no changes to the VFS data structures or interfaces. Modifications are limited to on-disk metadata and the in-memory file system specific fields of the VFS metadata. Ext3cow adds metadata to inodes/vnodes, directory entries, and the superblock that allows it to scope requests from any point-in-time into a specific file version and support scoping inheritance.

3.1 Superblock

Implementing snapshot requires some method of keeping track of the *snapshot epoch* of every file in the system. We place a system-wide *epoch counter*, stored in the on-disk and in-memory superblock, as a reference for marking versions of a file. The counter is a 32-bit unsigned integer, representing the number of seconds passed since the Epoch. Using one second granularity, the 32-bit counter allows us to represent approximately 132 years of snapshots. Our design includes the possibility of increasing the granularity of snapshots by adding additional bits to represent fractions of seconds. However, as the frequencies increase to sub-second snapshot, the significance of versions decreases and the usability of the interface suffers.

To capture a point-in-time image of a file system the superblock epoch number is updated atomically to the current

[1]In the future, we will be able to support files of the form `string1@string2` in which `string2` contains at least one non-numeric character.

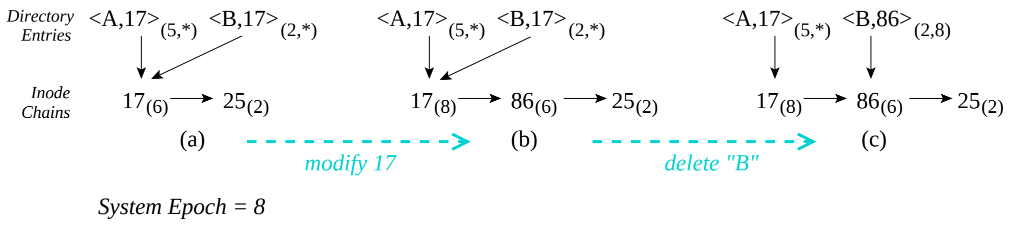


Figure 4: An example of names scoping to inodes over time.

system time. Creating new file versions is not done at the time of the snapshot, but, rather, at the next operation that modifies the data or metadata of an inode, *e.g.* a write, truncate, or attribute change. Snapshots may be triggered internally by the file system or by an `ioctl` call made by the user-space snapshot utility.

3.2 Inodes

Inode versions identify how a file's data and attributes have changed between snapshots. For each system-wide snapshot, a file may have an inode that describes it in that epoch. A file that has not changed during an epoch shares an inode with the previous epoch(s). While space is very tight in the 128 byte ext3 on-disk inode, we were able to squeeze in an additional 20 bytes of information by removing empty fields used for sector alignment, as well as fields to support the HURD operating system. Ext3cow does not currently support HURD. Future versions of ext3 may expand the inode size to 256 or 512 bytes, eliminating space constraints [47].

Three fields were added to both the on-disk and in-memory representation of the inode (Figure 3). A 32-bit `i_epochcounter` describes to which epoch an inode belongs. The `i_cowbitmap` maintains the block-versioning state of a file and is described in detail in Section 5.2. Lastly, we have added a pointer to the next version of an inode with the `i_nextinode` field.

3.3 Directory Entries and Naming

Directories in ext3, and therefore in ext3cow, are implemented as inodes in which the data blocks contain directory entries. Ext3cow versions directory inodes in the same manner as file inodes.

The directory entries themselves are versioned by adding scoping metadata to the directory entry (*dirent*). In ext3, a directory entry contains an inode number, a record length, a name length, and a name. To this, we add a *birth epoch* and a *death epoch* that determine the times during which a name is valid. Extending the directory entry is trivial and under no space constraints, because its length already varies in order to handle variable length names and name deletions.

Retaining names in ext3cow does not increase the size of directories. Ext3 removes names by increasing the record length of the preceding directory entry to span the deleted entry. The space for deleted names are not reused nor are directories compacted. Thus, keeping all names created over the lifetime of a file system does not increase the size of directories, and, therefore, does not decrease directory performance.

4 Version Scoping

Ext3cow maps point-in-time requests to snapshots and object versions through scoping metadata in directory entries and names. The logically continuous (to the second) time-shifting interface does not match exactly the realities of versioning. Several system properties govern ext3cow's versioning model. First, a version of file metadata or data covers a period of time; generally many different snapshot epochs. Also, ext3cow retains data at the time of a snapshot and does not track intermediate changes. When updating data or metadata, ext3cow marks versions with the current system epoch, not the current time. Finally, ext3cow maps point-in-time requests to the version preceding the exact time of the request. All told, this means that when accessing data in the past, all modifications that occur during an epoch are indivisible and occur at the start of an epoch.

A notable boundary case arises in the snapshot number returned by the `snapshot` utility (Section 2). Intuitively, the snapshot number should provide access to the file system at the time at which the snapshot was taken, which is exactly what it does. `Snapshot` returns the current time and sets the system epoch counter to this value plus one. The return value, say j, provides a handle to all changes included in the previous epoch. The system sets the counter for the current epoch to $j+1$. The next snapshot taken at k covers the period $[j+1,k]$. Access to any time in this interval, including k, retrieves data marked with epoch $j+1$.

Scoping backward in time provides a natural interface for file versioning and recovery. For example, a user accidentally deletes a file at time $t \in [j+1,k]$ but cannot remember when it was deleted. To restore the file, he or she recalls a time $s < t$ when the file exists and specifies s in the time-shifting interface.

4.1 Scoping Inodes

Inode chains provide a continuous-time view of all versions of a file. The chain links inodes backward in time. To find an inode for a particular epoch, ext3cow traverses the inode chain until it locates an inode with an epoch less than or equal to the requested point-in-time. At the head of the chain sits the most recent version of the inode. This design minimizes access latency to the current version – the most common operation. Figure 4(a) shows inode `17` last written during epoch `6`. Subsequent to that write, a snapshot has been taken, indicated by the system epoch counter value of `8`. A modification to inode `17` (Figure 4(b)) results in the inode being duplicated. Ext3cow allocates new inode `86` to which it copies the contents of inode `17`. Inode `86` is assigned epoch `6` and marked as unchangeable. Inode `17` is brought to the current epoch and remains a live, writable inode. Ext3cow's inode copy-on-write policy creates *stable inodes*, preserving a files inode number over the lifetime of a file.

Because of stable inodes, ext3cow supports the Network File System (NFS [24, 37]). NFS file handles are essential to its stateless operation and require the inode numbers to remain the same over the lifetime of a file handle.[2] Also, stable inodes preserves semantics for other applications, such as `ls -i` which prints inode numbers or applications that check inode numbers through `stat` to ensure that files have not been renamed.

4.2 Scoping Directory Entries

Directory entries are long lived, with a single name spanning many different versions of a file, each represented by a single inode. Figure 4 shows directory entries as a name, inode pair with the birth and death epoch as subscripts. The inode field points to the most recent inode to which the name applies. For example, name `A` points to inode `17` at the head of the inode chain. The name first occurred during epoch 5 and is currently live, represented by `*`. An `*` leaves live names open-ended so that as the inode continues to be versioned and the inode epoch increases, the directory entry remains valid. When removing a name, ext3cow updates the death epoch to indicate the point-in-time at which the name was removed. In Figure 4(c), name `B` dies and the death epoch is set to `8`. The name `B` is no longer visible in the present and will not be visible for any point-in-time request that scopes to snapshot epoch `8`.

The flexibility of birth/death epoch scoping respects the separation between names and inodes in UNIX-like file systems. Many names may link to a single inode. Also, a different number of names may link to an inode during different epochs. The same name may appear multiple times in the same directory, linking to different inodes during non-overlapping birth/death periods.

One concern with our scoping data structures is the linear growth of the inode chains over time. For frequently written files, each snapshot represents a new link in the chain and so accesses to versions in the distant past may be prohibitively expensive. While file systems have a history of linear search structures, *e.g.* directories in ext3, we find the situation unacceptable.

Ext3cow restricts version chains to a constant length through birth/death directory entry scoping.[3] When the length of a version chain meets a threshold value, ext3cow terminates that chain by setting the death epoch of the directory entry used to access this chain to the current system epoch and creates a new chain (of length one) by creating a duplicate directory entry with a birth epoch equal to the system epoch. The stability of inodes ensures that other directory entries linking to same data find the new chain. A long-lived, frequently-updated file is described by many short chains rather than a single long chain. While directory entries are also linear-search structures, this scheme increases search by a constant factor and will increase search exponentially when ext3 adopts directory indexing [47].

4.3 Temporal Vnodes

The piece-wise traversal of file system paths makes it difficult to inherit time scope along pathnames. For paths of the form `.../B@time/C...`, time-shifting specifies that `B`, and its successors, are accessed at `time`. When accessing `C`, the file system provides only `B's` inode as context. Because `time` rarely matches the epoch number of `B` exactly, `B`'s inode frequently has an epoch number prior to `time`. In this case, the exact scope is lost. For example,

[2]Note to reviewers: It is possible, albeit awkward, to make NFS work without stable inodes. We chose to simplify these statements to preserve the flow of the paper.

[3]Note to reviewers: this function will be implemented by the time of the conference.

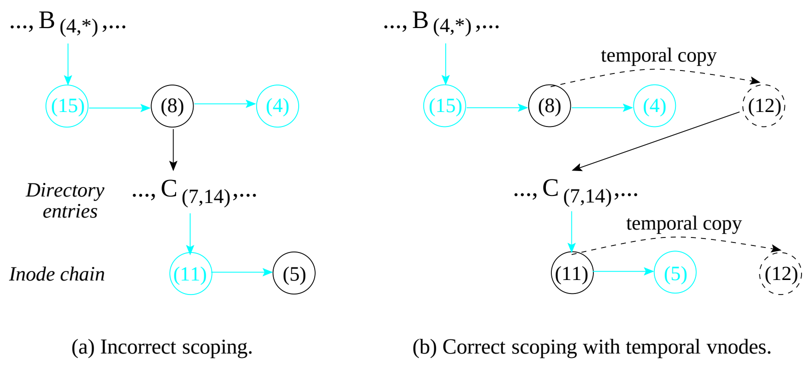


Figure 5: Accessing a path `...B@12/C...` in ext3cow. Directory entries are shown with birth and death epochs. Inodes (circles) are show with the epoch in which the inode was created. Inode numbers are not shown. Black directory entries and inodes indicate the access path according to scoping rules. The inode chain is traversed until an inode with creation epoch prior to the epoch of the parent inode is found. Temporal vnodes, in-memory copies of inodes, make this process accurate by preserving epoch information along access paths.

Figure 5(a) illustrates the wrong version of `C` being accessed. The access to `C` should resolve to the inode at epoch `11`, but leads mistakenly to the inode at epoch `5`.

To address this problem, ext3cow gives to each time context that accesses an inode a private in-memory inode (vnode) scoped exactly to the requested time. We call this a *temporal vnode* for two reasons: it is temporary and it implements time scoping inheritance. To make a temporal vnode, ext3cow creates an in-memory copy of the vnode to which the request scopes and sets the epoch number of the vnode to the requested time. It also changes the inode number to disambiguate the temporal vnode from the active vnode and other temporal vnodes. To avoid conflicts, the modified inode number lies outside of the range of inodes used by the file system. The temporal vnode correctly scopes accesses to directory entries (Figure 5(b)). This creates potentially many in-memory copies of the same inode data. Because data in the past are read-only, the copies do not present a consistency problem. The temporal vnode exists until the VFS evicts it from cache. Subsequent accesses to the same name (*e.g.* `B@12`) locates the temporal vnode in cache. Temporal vnodes are unchangeable and cannot be marked dirty.

Live inodes operate normally, concurrent or subsequent accesses in the present share a single copy of the vnode with the original inode number – corresponding to the inode on disk.

5 Versioning with Copy-on-Write

Ext3cow uses a *disk-oriented* copy-on-write scheme that supports file versioning without polluting Linux's page cache. Copies of data blocks exist only on disk and not in memory. This differs from other forms of copy-on-write used in operating systems that create two in-memory copies, such as process forking (`vfork` [22]) and the virtual memory management of shared pages. Ext3cow has the same memory footprint for data blocks as ext3, and, thus, does not incur overheads for copying pages or by using more memory, which reduces system cache performance.

Ext3cow employs the copy-on-write of file system blocks to implement multiple versions of data compactly. Scoping rules allow a single version of a file to span many epochs. Therefore, ext3cow needs to create a new physical version of a file only when data changes. Frequently, physical versions have much data in common. Copy-on-write allows versions to share a single copy of file system blocks for common data and have their own copy of blocks of data that have changed (Figure 6).

When the most recent version of a file precedes the system epoch in time, any change to that file creates a new physical version. The first step is to duplicate the inode, as discussed in Section 4. The duplicated inodes (new and old) initially share all data blocks in common. This includes sharing all indirect blocks, also doubly and triply indirect blocks. The first time that a logical block in the new file is updated, ext3cow allocates a new physical disk block to hold the data, preserving a copy of the old block for the old version. Subsequent updates to the same data in the same epoch are written in place; copy-on-write occurs at most once per epoch. Updates to data in indirect blocks (resp.

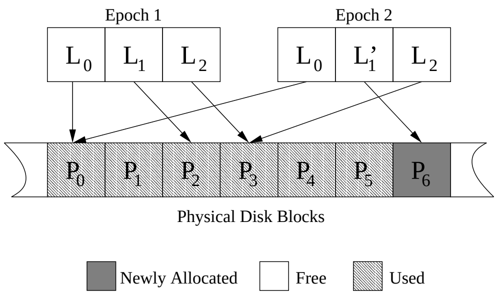


Figure 6: An example of copy-on-write. The version from epoch 2 updates logical block L_1 into L_1'. Ext3cow allocates a new physical (disk) block P_6 to record the difference. All other blocks are shared.

doubly and triply indirect blocks), change not only data blocks, but also indirect blocks. Ext3cow allocates a new disk block as a copy-on-write version of an indirect block.

5.1 Memory Management

To implement copy-on-write, we isolate the function to the on-disk file system; we do not trespass into kernel components such as the VFS or page cache. To achieve this isolation, ext3cow leverages Linux's multiple interfaces into memory; memory pages that hold file data are comprised of file system buffers, which map in-memory blocks to disk blocks. Ext3cow performs copy-on-write by re-allocating the file-system blocks that represent the storage for a buffer. In Linux, the VFS passes a `write` system call to the on-disk file system prior to updating a page in memory. This allows a file system to map the file offset to a memory address and bring data into the cache from disk as needed. In ext3cow, we take this opportunity to determine if a file block needs to be copy-on-written and to allocate a new backing block when necessary. Ext3cow replaces the disk block that backs (provides storage for) an existing block in the buffer and marks the buffer dirty. Then, the `write` call proceeds using the same memory page. At some point in the future, the buffer manager writes the dirtied blocks to disk as part of cache management. The actual copy is created at this time. Through re-allocation, ext3cow creates on-disk copies of file system blocks without copying data in memory.

The copy-on-write design preserves system cache performance and minimizes the I/O overheads associated with managing multiple versions. Ext3cow consumes no additional memory and does not pollute the page cache with additional data. Additionally, for data blocks, copy-on-write incurs no additional I/O, because the dirtied buffers are updated by the `write` call and need to be written back to disk anyway. The only deleterious effect of copy-on-write is I/O for indirect blocks, which do not necessarily get updated as part of a `write`.

5.2 Copy-on-write State bitmaps

Ext3cow embeds bitmaps in its inodes and indirect blocks that allow the system to record which blocks have had a copy-on-write performed. In the inode, ext3cow uses one bit for each direct block, one for the indirect, doubly-indirect, and triply-indirect block respectively. A bit of value 0 indicates that a new block needs to be allocated on the next write and bit value 1 indicates that a new allocation of this block has been performed within the current epoch and that data may be updated in place. Ext3cow zeroes the entire bitmap when duplicating an inode. Similarly, in an indirect block (resp. doubly or triply indirect block), the last eight 32-bit words of the block contain a bitmap with a bit for every block referenced in the indirect block. Ext3cow zeroes the bitmap when creating a copy-on-write version of the indirect block. Because bitmaps are cleared on new allocations, nothing needs to be done when snapshots are created. The bitmap design allows the bitmaps to be updated lazily – only when data are written.

Even though bitmaps borrow space in indirect blocks, our design preserves the maximum file size. The eight bitmap words reduce the number of blocks that a 1KB indirect block references from 256 to 248. This decreases the amount of data represented by the direct/indirect block tree from 16,843,020 blocks to 15,314,756 blocks. However, the reduced number remains larger than $2^{32}/\mathrm{BLOCKSIZE}$ addressing limit of a 32-bit processor. The same argument holds on 64-bit processors in which quadruply indirect blocks will be required for both ext3 and ext3cow.

The bitmap design allows ext3cow to improve performance when truncating a file. Truncate is a frequent file system operation: applications often truncate a file to zero length as a first step when rewriting that file. On truncate, ext3 deallocates all blocks of a file. In contrast, ext3cow deallocates only those blocks that have been written in

Operational Test	**ext3**	**ext3cow**
Test 1: Creates	46.67 ms	46.91 ms
Test 2: Removes	6.89 ms	3.54 ms
Test 3: Lookups	0.95 ms	0.95 ms
Test 4: Attributes	7.24 ms	7.38 ms
Test 5a: Writes	71.28 ms	71.91 ms
Test 5b: Reads	15.13 ms	15.11 ms
Test 6: Readdirs	20.28 ms	21.11 ms
Test 7: Renames	5.04 ms	6.57 ms
Test 8: Readlink	8.52 ms	18.57 ms
Test 9: Statfs	105.91 ms	105.22 ms

Table 1: Results from the "basic" tests of the Connectathon benchmark suite.

the current epoch. Other blocks remain allocated to be used in older versions of the file. Therefore, ext3cow skips deallocation for any blocks for which the corresponding state bitmap equals zero. For indirect blocks (resp. doubly or triply indirect blocks), ext3cow skips deallocation for the entire subtree underneath that block corresponding to the zero bit. In this way, ext3cow minimizes I/O to deallocate blocks and update free-space bitmaps during truncate.

6 Performance Evaluation

In order to quantify the cost/benefit trade-offs of versioning, we administered a variety of experiments comparing ext3cow to its sister file system – unmodified ext3. Experiments were conducted on an IBM x330 series server, running RedHat Linux 7.3 with the 2.4.19 SMP kernel. The machine is outfitted with dual 1.3 GHz Pentium III processors, 1.25 GB of RAM, and an IBM Ultra2 18.2G, 10K RPM SCSI drive. Experiments for both ext3cow and ext3 were performed on the same 5.8 GB partition.

6.1 Micro-benchmarks

The Connectathon NFS test suite evaluates operational correctness and measures performance. There are nine parts to the "basic" series of tests. Each part tests a separate system call. In order, they are: (1) create 155 files 62 directories 5 levels deep, (2) remove these files, (3) 150 getcwd calls, (4) 1000 chmods and stats, (5) write a 1048576 byte file 10 times and read it 10 times, (6) create and read 200 files in a directory using `readdir`, (7) create ten files, rename and stat both the new and old names, (8) create and read 10 symlinks, and, lastly, (9) perform 1500 statfs calls.

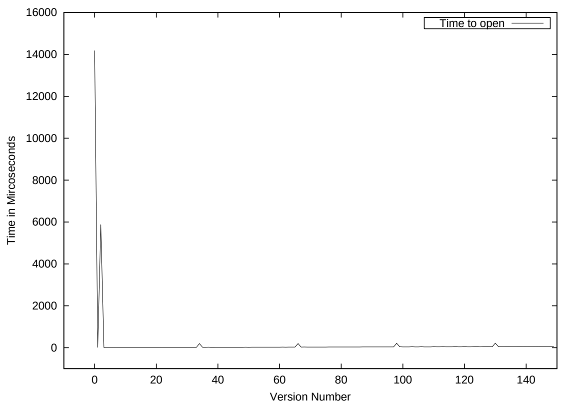
The results of the Connectathon basic test average the results of 10 runs on a newly mounted (cold cache) file system. Ext3cow meets or exceeds the performance of ext3 in almost all areas.Table 1 shows the average cumulative time to perform each test.

Ext3cow outperforms ext3 on tests that deallocate inodes of data. Test 2 (Removes) is one example. Both systems update a directory entry: ext3cow sets the death epoch and ext3 removes the name by updating record length of the adjacent directory entry. However, only ext3 updates inode bitmaps to deallocate inodes for removed files. Ext3cow never deallocates inodes or data. We expect similar savings on on microbenchmarks that truncate or clear portions of a file. Connectathon does not test these functions.

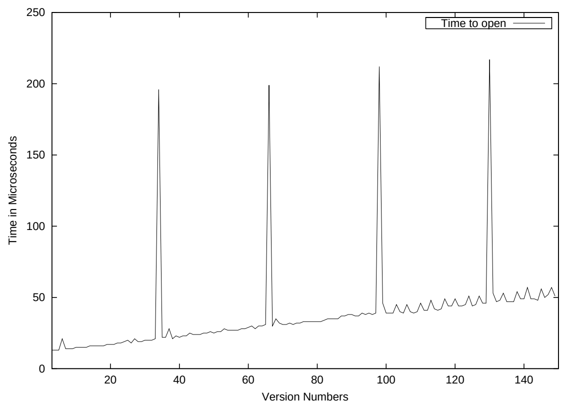
String operations to support versioning result in ext3cow under-performing ext3 on tests dominated by name operations. During lookup, ext3cow parses every name looking for the `@` version specifier. Ext3cow takes the string prior to `@` as a file name and uses the remainder of the string for scoping. It performs similar string parsing when reading symbolic links. Tests 7 (Renames) and 9 (Readlinks) show string manipulation overhead. Test 3 (Lookups) does not have this overhead, because it does not call the on-disk file system lookup. Rather, test 3 calls the VFS entry point `getcwd`, which can be satisfied out of the VFS's directory entry cache.

6.1.1 Performance in the Past

To capture the effect of multiple versions on performance, we modified the Connectathon benchmark to measure the time to open a series of 150 versions of a file from youngest to oldest. These versioned inodes were created consecutively and, therefore, ext3cow lays them out near-contiguously in block groups. Figure 7(a) shows the results



(a) Opening Past Inodes



(b) Opening Past Inodes (Enlarged)

Figure 7: The time to open 150 versions of a file in a best-case example.

of this test on a cold cache. To access the first inode, the system incurs two disk seek penalties for I/O: one to lookup the inode by name and one to access the inode. Almost all subsequent inode accesses are served out of different caches. Figure 7(b) shows a closeup of Figure 7(a) with large values filtered out. The baseline represents fetches out of the file-system cache, with linear scaling because accessing the k^{th} version traverses k inodes in cache. Every 32 inodes, the file system fetches a new group (4KB) of inodes from disk, which based on the low-latency, seem to be served out of the disk's cache. Having fetched the next group of inodes, subsequent accesses to these inodes are served in the file system cache.

By allowing inodes to be cached after their first access, these experiments show the sub-structure of versioning. Experiments that flush the cache between accesses to the k^{th} and $k+1^{th}$ versions give a more accurate estimate of total time to fetch an inode in the past. However, all versions appear to take 14 ms, because I/O dominates in-memory operations.

We also attempted worst-case versions of this experiment by artificially aging a file system. Between successive inodes in a chain, a block group worth of other inodes (¿16,000) were created. The goal was to subvert and render ineffective ext3cow's inode clustering. The results of these experiments are indistinguishable from the best-case experiments, showing an initial penalty for I/O and subsequent access out of caches. Worst-case experiments do not show the regular spikes every 32 inodes seen in the best-case. The message of worst-case trials is that accessing data in the past grows linearly in the number of versions, but the linear growth is dominated by and indistinguishable from I/O penalties to read the first inode in the chain.

6.2 Trace-driven Experiments

To examine the effect of snapshot on metadata allocation, we used three months of system call traces from Berkeley [35] to populate a file system partition, and performed an off-line analysis to identify the type and amount of allocation. By aging a file system, we more accurately measure and analyze real-world performance [42]. The traces were played back through two file systems: ext3, used as a baseline for comparison and ext3cow. In ext3cow, we used three policies to quantify the allocation difference for various snapshot frequencies. Snapshots were taken at 24 hour, 1 hour and 1 minute intervals. The traces contain no data or name hierarchy information, so data was synthesized as needed and a hierarchy structure was built by `chdir` and `mkdir` calls.

Table 2 displays a 4.1% increase in metadata for 24 hour snapshots and a 4.8% increase for 1 minute snapshots. These results indicate an initial jump in the amount of metadata to support any amount of versioning, followed by more gradual growth as snapshot frequencies increase. These results are consistent with those presented by Soules, *et. al* [43]. Of the 4.8% increase in metadata 1.8% comes from versioned directory inodes. A change in design to reduce versioned directory inodes may have a significant impact on allocation and performance (see Section 8.5.).

File System	Allocated Inodes	Dir Inodes
ext3	78625	1386
ext3cow – 24 hour	81858 (+3233)	2264 (+878)
ext3cow – 1 hour	81936 (+3311)	2349 (+963)
ext3cow – 1 min	82435 (+3810)	2248 (+1062)

Table 2: The total number of allocated inodes and the number of those inodes allocated for directories for the ext3 and ext3cow file systems over various snapshot frequencies.

6.3 The Andrew File System Benchmark

The Andrew File System Benchmark [18, 27] is a research and industry standard benchmark design to artificially reproduce and measure the actions of a typical UNIX user. The benchmark executes five phases of measurements: creating and traversing large numbers of directories, transferring large amount of data, examining the status of the metadata of many files, reading the data from many files, and compiling and linking a large program. The benchmark ran on the ext3 partition in 1.90 seconds, and took 1.89 seconds on ext3cow. Because no snapshots were taken during the trial, no extra data or metadata were consumed by ext3cow.

7 Related Work

The Cedar file system [12, 15, 40] is the first example of a file system that maintains versions of a file over time. Versions were shared among file system users. Each write operation creates a new version where each version has a unique name to the file system, *e.g.* /home/user/ext3cow.tex!3 represents the third version. Each version of a file is autonomous, with no shared data between versions; sharing a file requires transferring all blocks of a file. A similar approach to versioning was taken in VMS [7, 21], and TOPS [26].

Other file systems, such as AFS [18, 25], LFS [36, 41], Plan-9 [31, 32], Spiralog [13, 20], WAFL [17] and SnapMirror [29], use snapshot and versioning for file system recovery. None of these implementation provide a transparent interface to access files, nor do they allow for the browsing or linking of old versions. A survey and evaluation of snapshot and backup techniques was performed by Chervenak *et. al* [5] and Azagury *et. al* [1].

The Elephant file system [38, 39] is the first to include a variety of user-specified retention policies as well as an intuitive, date-oriented interface. The retention policies may be intrusive to the user. Particularly when compared with user-space version-control system such as RCS [34, 46] or CVS [9, 14].

In the comprehensive versioning file system (CVFS) [43], all writes to the server in a client/server storage system are versioned. Versions are accessed by mounting a point-in-time view of the file system. Because CVFS operates on a file server, it sets up a performance/feature trade-off based on whether clients use write caching. With write caching on, CVFS precludes the versioning of files that are overwritten in client caches before being written to storage or, with caching off, it eliminates the performance benefits of asynchronous I/O.

8 Project Status and Future Work

In the spirit of open source software distribution, we have released the first version of ext3cow to the public via [omitted]. At this time, ext3cow has had hundreds of visitors and tens of downloads. A development mailing list has been created, to which a number of enthusiasts have subscribed. The authors have been running ext3cow to store data on their laptops and personal workstations since June 2003. We have not experienced a system crash or data loss incident in that period.

The development of ext3cow has left us with features yet to be implemented and open research questions. Many issues revolve around space reclamation in file systems that never delete data.

8.1 Fine-grained Versioning

The design of ext3cow permits for fine-grained snapshots of files, directories, and directory subtrees. Currently, ext3cow uses a system-wide epoch to evaluate whether new versions of files and directories need to be created. To implement fine-grained versioning, ext3cow would add an additional field, a snapshot epoch, to each inode. Snapshots may then be taken on individual files or directories, by updating the snapshot epoch of an individual inode, rather than the system epoch. Snapshots on directories would snapshot recursively the namespace under that point. Copy-on-write and inode duplication would be evaluated bottom up, based on the snapshot epoch of the current inode followed by the parent inode, and up the parent chain to the file system root.

This approach has semantics advantages and performance disadvantages. It allows portions of the namespace to be versioned independently. Furthermore, it allows for files to be versioned on a per-session (open/close) basis; file snapshots are taken implicitly by the file system on open. The disadvantage lies in evaluating scope, for which the whole path to the root needs to be examined for copy-on-write and inode duplication. Intermediate strategies that version files and directories without recursion are possible.

Ext3cow does not support versioning on every write, which has been proposed as an intrusion recovery mechanism for file system servers [43]. While attractive in the client/server model, tracking every write does not translate well to file systems for individual computers. In these architectures, multiple writes are performed in memory outside of the file system's control – in process buffers and in the shared page cache for memory mapped files. Each write to the file system may encapsulate a large number of "writes" from processes into the cache. Therefore, versioning every update cannot be done by a disk file system. Rather, it would require memory system integration along the lines of I/O-lite [28].

8.2 Same-epoch Deletion

The literature shows that most files are short lived; often, they are deleted within seconds after their creation [2, 11, 35]. This begs the question – how important are these files to users, and what is their relevance in a versioning file system? Currently, ext3cow retains these files. However, we know of no studies on the temporal significance of files that justify this decision.

There are benefits to permanently removing files that are unlinked in the same epoch in which they are created. Short-lived files consume disk space in ext3cow. We are not so concerned with the space usage from a capacity standpoint. We are more concerned with the negative effects that useless data has on the locality of files on disk. Data in short-lived files spreads all other data across the disk, leading to longer seeks.

Implementing same-epoch deletion in ext3cow is trivial. The difficulty lies in evaluating trade-offs between the importance of this data and the negative effects its retention has on a system.

8.3 Placement Policies

The issue of how to place data to optimize read performance in versioning systems remains mostly unexplored. Many versioning systems are log-structured [13, 36, 41] or write-optimized [17], and do not attempt to organize storage for read. Versioning systems built on read-optimized file systems have largely ignored the issue of placement [33, 38, 39, 43].

Versioning works against read-optimizations in file systems. FFS, ext3, and like file systems use several techniques to improve read performance. Related inodes are clustered on disk to minimize seek time and improve the effectiveness of disk-based caching and read-ahead [10, 23]. Also, file data are placed contiguously to improve sequential read performance. By creating duplicate inodes, ext3cow reduces the efficacy of inode clustering. Fine-grained copy-on-write data blocks destroy contiguity; *e.g.* for file versions that share data blocks, not all versions can be contiguous at the same time. Currently, ext3cow places the first version of file data contiguously and contiguity erodes over time.

We have conceived of and experimented with initial strategies for maintaining read performance. For file data, we have developed the concept of *virtual contiguity* [30] in which strict contiguity is relaxed and versions of blocks are be placed "near each" other on disk. The goal is to densely populate a small region of disk with multiple versions of the same block. Ext3cow will read this small region in a single I/O, eliminating seeks between non-contiguous blocks. Data not needed in the current version are discarded by the file system. Blocks that are out-of-order on disk pose no problem in memory, because they are logically rearranged with pointers – no copying of data is necessary. Virtual contiguity promises to make disk-based caching and read-ahead work better. Read policies should provide near-contiguous performance for all versions of a file. For inodes, we are experimenting with policies that rearrange existing inode placements to preserve clustering. Ext3cow could clip old segments of an inode chain and relocate them to a less populated regions of storage. This will free space for new inode allocations near the desired cluster. Both placement policies are sensitive to parameterizations and require further investigation.

8.4 Snapshot Reclamation

Although disk is relatively inexpensive, reducing the storage footprint of a file system benefits management and performance. Both metadata and data may be recovered through the reclamation of snapshots. The time-shifting interface may be extended to allow all data prior to a time to be discarded, or, to compress all snapshots between two points in time to a single snapshot. The technical challenge of snapshot reclamation lies in resolving the data blocks and inodes that belong to a given time period. For example, to deallocate a data block of an inode, future and past inodes in the inode chain must be compared to ensure that the data block is not shared by other versions. Fine-grained copy-on-write and the longevity of scope leads to inodes and data being shared among many snapshots.

8.5 Directory Compression

In ext3 and ext3cow, inodes are a fixed and limited resource. Although our findings show only a modest increase in metadata, further investigation into saving inodes may be warranted. Results show that 64% of versioning inode overhead comes from directory inodes. In our current design, directory inodes are duplicated on modification after a snapshot just like file inodes. However, this is not strictly necessary when the only changes between inode versions is the addition or removal of directory entries. Because directory entries have scoping information built in, all directories share the same direct and indirect blocks. The only reason to duplicate inodes is to store a snapshot epoch number for scoping.

In an alternate design, we may choose to interleave *epoch entries* with directory entries in the directory inodes direct and indirect blocks. In place of allocating and chaining a new inode for each update after a snapshot, an epoch entry would be placed in-line with the inode's directory entries. A related design based on logging, with commensurate benefits, was implemented in CVFS [43].

9 Conclusions

Ext3cow is a fully implemented open-source file system that provides users with a new and intuitive interface for accessing data in the past. Ext3cow's versioning interface supports many features: easy access to on-line backups; a way to detect and recover from system tampering; read-only point-in-time snapshots for data mining; and, file-oriented deletion recovery. To provide these functions, ext3cow uses a copy-on-write scheme and versioning metadata that incur little overhead and exhibit a small data footprint. All modifications made to ext3cow are encapsulated within the on-disk file system, avoiding the disadvantages of a VFS or memory manager implementations.

References

[1] A. Azagury, M. E. Factor, and J. Satran. Point-in-time copy: Yesterday, today and tomorrow. In *Proceedings of the Tenth Goddard Conference on Mass Storage Systems and Technologies*, pages 259–270, April 2002.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems (SOSP)*, October 1991.

[3] R. Bryant, R. Forester, and J. Hawkes. Filesystem performance and scalability in Linux 2.4.17. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 259–274, June 2002.

[4] R. Card, T. Y. Ts'o, and S. Tweedie. Design and implementation of the second extended file system. In *Proceedings of the 1994 Amsterdam Linux Conference*, 1994.

[5] A. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings of the Joint NASA and IEEE Mass Storage Conference*, March 1998.

[6] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the 1992 Winter USENIX Technical Conference*, pages 43–60, 1992.

[7] Digital Equipment Corporation. *Vax/VMS System Software Handbook*, 1985.

[8] EMC Corporation. *EMC TimeFinder Product Description Guide*, 1998.

[9] P. Cederqvist *et. al*. *Version Management with CVS.* Network Theory Limited, 2003. http://www.network-theory.co.uk/cvs/manual/.

[10] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17, January 1997.

[11] T. Gibson, E. Miller, and D. Long. Long-term file system activity and interreference patterns. In *Proceedings of the 23rd Annual International Conference on Computer Measurement and Performance*, pages 976–987, December 1998.

[12] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, March 1988.

[13] 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.

[14] D. Grune, B. Berliner, and J. Polk. Concurrent versioning system (CVS). http://www.cvshome.org/, 2003.

[15] R. Hagman. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating systems principles (SOSP)*, pages 155–162, 1987.

[16] Hitachi, Ltd. *Hitachi ShadowImage*, June 2001.

[17] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the USENIX San Francisco 1994 Winter Conference*, January 1994.

[18] 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.

[19] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. In *3rd Symposium on Operating System Design and Implementation Proceedings (OSDI)*, pages 239–250, February 1999.

[20] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, 6(1):51–81, 1996.

[21] K. McCoy. *VMS File System Internals*. Digital Press, 1990.

[22] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

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

[24] Sun Microsystems. *NFS: Network file system protocol specification*. Network Working Group, Request for Comments (RFC 1094), March 1989. Version 2.

[25] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[26] L. Moses. An introductory guide to TOPS-20. Technical Report TM-82-22, USC/Information Sciences Institute, 1982.

[27] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer Conference Proceedings*, pages 247–256, 1990.

[28] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, Feb 2000.

[29] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, pages 117–129, Jan 2002.

[30] Z. N. J. Peterson. Data placement for copy-on-write using virtual contiguity. Master's thesis, University of California, Santa Cruz, September 2002.

[31] D. Presotto. Plan 9. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 31–38, April 1992.

[32] S. Quinlan. A cached worm file system. *Software – Practice and Experience*, 21(12):1289–1299, Dec 1991.

[33] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the 2002 Conference on File And Storage Technologies (FAST)*, pages 89–101, January 2002.

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

[35] D. Roselli and T. E. Anderson. Characteristics of file system workloads. Research report, University of California, Berkeley, June 1996.

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

[37] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Summer USENIX Conference Proceedings*, pages 119–130, June 1985.

[38] D. J. Santry, M. J. Feeley, N. C Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2–7, 1999.

[39] D. J. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 110–123, December 1999.

[40] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP)*, pages 25–34, 1985.

[41] M. Seltzer, K. Bostic, M. K. McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Conference, San Diego, CA, USA, 25-29 January 1993*, pages 307–326, January 1993.

[42] K .A. Smith and M. I. Seltzer. File system aging – Increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS Conference*, pages 203–213, June 1997.

[43] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, March 2003.

[44] J. D. Strunk, M. L. Scheinholtz G. R. Goodson, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, October 2000.

[45] A. S. Tannenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall Inc., Englewood Cliffs, NJ 07632, 1987.

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

[47] T. Y. Ts'o and S. Tweedie. Planned extensions to the Linux ext2/ext3 filesystem. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 235–243, June 2002.