Ext3cow: A Time-Shifting File System for Regulatory Compliance

ZACHARY N. J. PETERSON
and
RANDAL BURNS
Hopkins Storage Systems Lab
Department of Computer Science
Johns Hopkins University

The ext3cow file system, built on the popular ext3 file system, provides an open-source file versioning and snapshot platform for compliance with the versioning and audtitability requirements of recent electronic record retention legislation. Ext3cow provides a *time-shifting* interface that permits a real-time and continuous view of data in the past. Time-shifting does not pollute the file system namespace nor require snapshots to be mounted as a separate file system. Further, ext3cow is implemented entirely in the file system space and, therefore, does not modify kernel interfaces or change the operation of other file systems. Ext3cow takes advantage of the fine-grained control of on-disk and in-memory data available only to a file system, resulting in minimal degradation of performance and functionality. Experimental results confirm this hypothesis; ext3cow performs comparably to ext3 on many benchmarks and on trace-driven experiments.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*Directory Structure, File organization*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; E.5 [**Files**]: Backup recovery; H.3.2 [**Information Storage and Retrieval**]: Infomation Storage—*File organization*

General Terms: Design, Measurement, Performance, Reliability

Additional Key Words and Phrases: Versioning file systems, Copy-on-write

1. INTRODUCTION

Recent federal, state and local legislation have created new requirements for how to retain and access electronic information. There exist over 4,000 acts and regulations that govern digital storage, all with a varying range of requirements for maintaining electronic records. Examples include the Health Insurance Portability and Accountability Act (HIPAA) of 1996, the Gramm-Leach-Bliley Act (GLBA) of 1999, and the more recent Federal Information Security Management Act (FISMA) and Sarbanes-Oxley Act (SOX) of 2002. All

This work was supported in part by NSF award CCF-0238305, by DOE Office of Science award P020685, and by the IBM Corporation.
Author's address: Zachary Peterson, 224 New Engineering Bldg., 3400 N. Charles St., Baltimore, MD 21218. zachary@cs.jhu.edu

of this legislation requires an auditable trail of changes made to electronic records that are accessible in real-time. This implies versioning files and providing a means of quickly retrieving versions from any point in time.

To address the versioning and auditability needs of regulated storage, we have developed *ext3cow*, an open-source disk file system based on the third extended file system (ext3). Ext3 [Card et al. 1994] is the Linux default file system based on the Minix file system [Tannenbaum 1987] and influenced by the Fast File System (FFS) [McKusick et al. 1984]. Ext3 has become robust and reliable, providing reasonable performance and scalability for many users and workloads [Bryant et al. 2002]. Ext3cow extends the ext3 design by retrofitting the in-memory and on-disk metadata structures to support lightweight, logical file systems snapshots and individual file versioning. All files and snapshots are available at all times, and ext3cow offers a fine-grained user-interface to access individual file and directory versions from snapshots.

Ext3cow differs from other efforts at versioning file systems in its combination of features. Ext3cow both (1) encapsulates all versioning function within the on-disk file systems and (2) provides a fine-grained, interactive, and continuous-time interface to file versions and snapshots. We accomplish this through the *time-shifting* interface, which allows users and applications to interact directly with the disk file system, *i.e.* the interface is transparent to kernel components, in particular, the virtual file system. Other file systems that provide fine-grained access to versions do so by modifying kernel interfaces [Cornell et al. 2004; Muniswamy-Reddy et al. 2004; Santry et al. 1999; Santry et al. 1999]. This incurs copying overheads, pollutes the buffer pool with old data, and complicates installation and management. Other disk file systems provide coarse-grained access to versions through the creation of namespace tunnels [Hitz et al. 1994] or via mounting separate logical volumes [Strunk et al. 2000; Soules et al. 2003]. Some disk file systems provide no interface to versions, restricting versioning to internal use only [Rosenblum and Ousterhout 1992; Seltzer et al. 1993].

In time-shifting, ext3cow introduces an interface to versioning that presents a continuous view of time. Users or applications specify a file name and any point in time, which ext3cow scopes to the appropriate snapshot or file version. The time-shifting interface allows user-space tools to create snapshots and access versions. Applications may access these tools to coordinate snapshots with application state. User-space tools are suitable for automation, using software as simple as `cron`. Furthermore, snapshots fit well into an information lifecycle management (ILM) framework. ILM is a policy-based scheme for managing the lifetime of electronic information, including time-sensitive data migration and consolidation, backups and restoration, disaster recovery, and long-term archiving. Ext3cow's time-shifting and controlled versioning facilitates the consistent transfer of data from ext3cow to other storage systems.

Many of the virtues of ext3cow lie in encapsulating snapshot and versioning entirely within the on-disk file system. Ext3cow does not change any kernel interfaces and does not modify the common file object model provided by the virtual file system (VFS) [Kleiman 1986]. This makes ext3cow easy to install in existing systems; it may be loaded as a module to a running kernel and co-exist with all other Linux file systems. Only an on-disk file system, such as ext3cow, can control data placement, metadata organization, and I/O. Specifically, ext3cow retains tight control on the versioning of buffers and pages. Ext3cow does not degrade cache performance by insuring the Linux page cache sees a

single copy of file data; old versions of data exist only on disk. Copies are created on-demand when performing I/O to the disk. This is not possible in VFS implementations. Further, ext3cow's inode versioning policy maintains *stable inodes*, preserving a files inode number over the lifetime of a file. Because of stable inodes, ext3cow implicitly supports the Network File System (NFS [Sandberg et al. 1985; Microsystems 1989]). 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. Again, this is not possible in VFS implementations.

Lastly, some versioning systems require specialized, and often expensive hardware, making these systems unattractive for the consumer. Regulatory compliance places a tremendous financial burden on organizations. AMR research estimates the total spending on Sarbanes-Oxley compliance alone in 2004 to exceed $5 billion [Hagerty 2004]. Experience with HIPAA [P. Killbridge 2003] indicates that the costs of compliance are relatively greater for smaller organizations. This research is a key component in reducing the cost of compliance for small organizations. By providing an open-source system that satisfies the requirements of many electronic record management regulations, ext3cow will be particularly helpful to non-profits subject to government reporting requirements, small businesses subject to Sarbanes-Oxley, and small health care providers subject to HIPAA.

We have released ext3cow under the GNU Public License via *http://www.ext3cow.com*. As of this writing, ext3cow has had over a thousand visitors and hundreds of downloads from over one hundred different countries. We run a development mailing list 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. Ext3cow has appeal beyond the regulatory environment for which it is designed; it has been adopted as the storage platform for several research projects.

In the remainder of this paper, we present the details of the time-shifting interface and describe how file system data structures were retrofitted to support ext3cow's feature set in a disk file system. We present benchmarks and trace-driven experiments that show that versioning has a minor effect on the file system performance. On most micro-benchmarks, ext3cow meets the performance of ext3.

2. RELATED WORK

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 [Gifford et al. 1988; Hagman 1987; Howard et al. 1988; Chutani et al. 1992] and for archiving and data mining [Quinlan and Dorward 2002]. File versioning, creating new logical versions on every disk write or on every open/close session, is used for tamper-resistant storage [Soules et al. 2003; Strunk et al. 2000] and file-oriented recovery from deletion [Cornell et al. 2004; Muniswamy-Reddy et al. 2004; Santry et al. 1999]. Both techniques speed recovery and limit exposure to data losses during file system failure [Hitz et al. 1994; Seltzer et al. 1993]. A range of snapshot implementations exist, both at the logical file system level [Gifford et al. 1988; Howard et al. 1988; Hutchinson et al. 1999; Quinlan and Dorward 2002; Santry et al. 1999] and the disk storage level [EMC Corporation 1998; Hitachi, Ltd. 2001; Hutchinson et al. 1999;

Strunk et al. 2000].

File system versioning and snapshot have been used to recover from failure. FFS [McKusick et al. 1984; McKusick and Ganger 1999; McKusick 2002] takes snapshots to create a quiescent file system on which to perform on-line file system integrity checking. Writes that occur during a check are logged to a special snapshot file to which file system blocks are copy-on-written. FFS does not provide an interface to access file snapshots on-line. WAFL [Hitz et al. 1994] also uses snapshots for recovery. In addition, it provides users a `.snapshot` directory for every directory in the file system containing discrete views of the past.

File system snapshots implemented with copy-on-write are an implicit feature of log-structured file systems. LFS [Rosenblum and Ousterhout 1992; Seltzer et al. 1993] and Spiralog [Green et al. 1996; Johnson and Laing 1996] do not overwrite file data as they are written, but instead write changes as they occur to a circular log. Checkpoints, which serve as snapshots in log-structured file systems, are used to roll-back a file system to a known consistent point after a system failure. LFS and Spiralog do not provide an interface to access versions.

The Andrew file system [Howard et al. 1988; Morris et al. 1986], the Episode file system [Chutani et al. 1992], Plan-9 [Presotto 1992; Quinlan 1991], and SnapMirror [Patterson et al. 2002] use snapshot with copy-on-write techniques as a method to perform quick, low-bandwidth backups in an on-line fashion. Venti [Quinlan and Dorward 2002] uses hashing and copy-on-write to archive blocks efficiently. A survey and evaluation of snapshot and backup techniques was performed by Chervenak *et al* [Chervenak et al. 1998] and Azagury *et al* [Azagury et al. 2002].

Cedar [Gifford et al. 1988; Hagman 1987; Schroeder et al. 1985] is the first example of a file system that maintains versions of a file over time. Versions are shared among file system users. Each write operation creates a new version that 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. Similar approaches were used by VMS [Corporation 1985; McCoy 1990] and TOPS [Moses 1982].

The Elephant file system [Santry et al. 1999; Santry et al. 1999] is the first file system to include a variety of user-specified retention policies similar to user-space version control tools such as RCS [Rochkind 1975; Tichy 1985], PRCS [MacDonald et al. 1998] and CVS [Cederqvist *et al* 2003; Grune et al. 2003]. Elephant attempts to make intelligent decisions about which versions to keep, an approach taken by some on-line configuration management tools like CPCMS [Shaprio and Vanderburgh 2002]. Elephant provides an intuitive, date-oriented interface. It is implemented as a replacement for the BSD VFS layer and provides versioning to all on-disk file systems that support its interface. Wayback [Cornell et al. 2004] uses a similar versioning paradigm.

In the Comprehensive Versioning File System (CVFS) [Soules et al. 2003], all writes to the server, in a client/server storage system, are versioned, which provides an audit trail for security breaches. CVFS exists as a complete system, in which versions are accessed by mounting a point-in-time view of the file system over NFS [Sandberg et al. 1985; Microsystems 1989].

To place our contributions in context with respect to recent versioning file system re-

[1]WAFL is implemented as a file-system appliance within a custom operating system.

	ext3cow	CVFS	Elephant	Wayback	WAFL	LFS
Disk file system	•				•	•
Preserves kernel interfaces	•			•	N/A[1]	•
Files system snapshot	•	•			•	•
File versioning	•	•	•	•		
Time-oriented interface	•		•			
Preserves FS namespace	•	•	•			•
Stable inodes for NFS	•	•			•	•
Open-source license	•			•		•

Table I. Feature comparison of versioning file systems.

search, Table I compares the features of ext3cow to CVFS [Soules et al. 2003], Elephant [Santry et al. 1999], Wayback [Cornell et al. 2004], WAFL [Hitz et al. 1994], and log-structured file systems (LFS) [Rosenblum and Ousterhout 1992; Seltzer et al. 1993]. We restrict this treatment to file systems, omitting versioning archives [Quinlan and Dorward 2002; Batten et al. 2002], because we are concerned with interactive versioning in the regulatory environment. We also omit VersionFS [Muniswamy-Reddy et al. 2004] because it compares similarly to Wayback. This table punctuates our contribution. Ext3cow provides the benefits of fine-grained versioning with interactive, real time access to versions, without manipulating kernel interfaces. Table I also indicates that log-structured file systems provide an attractive alternative; ext3cow's features could be achieved by adding the time-shifting interface to an LFS. However, one of our principal goals in building ext3cow for the regulatory environment is secure deletion (Section 8), which obviates the use of the log-structured layout. The write policy of LFS spreads data from a single file throughout the log. The efficiency of our secure deletion architecture [Peterson et al. 2004] relies on the file system clustering data and metadata (indirect blocks) so that a small amount of secure overwriting [Gutmann 1996] securely deletes a large amount of data. We chose ext3 as our starting point, because it provides such clustering.

3. 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 in the regulatory environment. Some require old data to be accessed through a separate mount point [EMC Corporation 1998; Soules et al. 2003; Strunk et al. 2000], 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 [Corporation 1985; Gifford et al. 1988; Hagman 1987; McCoy 1990; Moses 1982], *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. WAFL [Hitz et al. 1994] uses namespace tunnels from the present to the past, 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

```
[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.
```

Fig. 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
```

Fig. 2. Creating distinguished (named) snapshots in ext3cow.

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 the kernel.

We designed the time-shifting interface for applications and enhance its interactive usability through shell extensions. Seconds since the Epoch conforms to `gettimeofday` and is the natural way for applications to query, store, and encode time. However, humans prefer richer time formats, such as `[[[[[cc]yy]mm]dd]hh]mm[.ss]` in the `date` utility. To enhance usability for humans, we are developing shell extensions in a *time-traveling* csh (ttcsh), which will support a variety of date, time and naming formats to help users browse versions (Section 8).

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.

As an interface, time shifting is useful but not complete. We do not wish to require users to remember when they created versions. Thus, we allow users to tag or enumerate all versions of a file, reporting versions and their scope (creation and replacement time).

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. Currently, the use of seconds since the limits (named) snapshots to one per second. For systems that use snapshot as part of recovery [Hitz et al. 1994], 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. Furthermore, upcoming support for microsecond granularity time in ext3 will remove all limitations on the frequency of snapshots in ext3cow.

4. 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 [Cornell et al. 2004; Muniswamy-Reddy et al. 2004; Santry et al. 1999; Santry et al. 1999; Zadok and Bădulescu 1999], 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, 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.

4.1 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. We choose the same representation of time as does ext3. When ext3 adopts microsecond granularity times, ext3cow will be able to represent arbitrarily fine-grained epochs in the superblock.

To capture a point-in-time image of a file system the superblock epoch number is updated atomically to the current 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 through the user-space snapshot utility.

4.2 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.

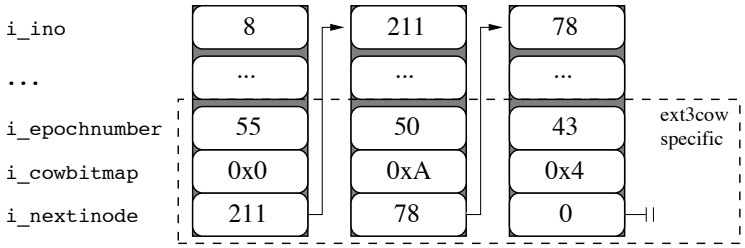


Fig. 3. Both on-disk and in-memory inodes were retrofitted to support snapshot and copy-on-write by adding three fields: an inode epoch number, a copy-on-write bitmap, and field pointing to the next inode in the version chain.

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 disk sector alignment, as well as fields for the HURD operating system, which is not currently supported. Future versions of ext3 will expand the inode size to 256 or 512 bytes, eliminating all practical space constraints [Ts'o and Tweedie 2002].

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. When writing data to an inode, the system updates the `i_epochcounter` to the system epoch counter. The `i_cowbitmap` maintains the block-versioning state of a file and is described in detail in Section 6.2. Lastly, we have added a pointer to the next version of an inode with the `i_nextinode` field.

Ext3cow supports both system-wide snapshots and individual file versions, by allowing a snapshot to be taken on a per-file basis. A variant of the snapshot utility, which takes a name as an argument, sets a file epoch to the current time. An individually versioned inode has the property that its `i_epochcounter` exceeds the system-wide epoch. On write, ext3cow detects this condition and performs copy-on-write based on the file's epoch rather than the system epoch. A subsequent system-wide snapshot, ends this condition and creates another new copy-on-write version of the file.

4.3 Directory Entries and Naming

Directories in ext3 and 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 names and name deletions. Because directory entries are scoped to an epoch range, names that have been unlinked, and, therefore, given a death epoch, may be reused in a future context to represent a new file. Ext3cow only deletes one class of files; it permanently removes files that are unlinked in the same epoch in which they were created.

Retaining file names in ext3cow does not increase the directory size when compared with ext3. Both systems unlink names by increasing the record length of the preceding

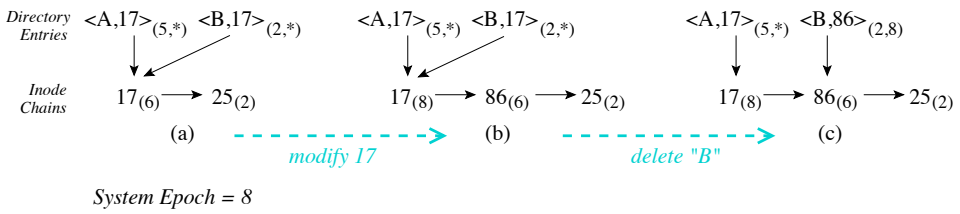


Fig. 4. An example of names scoping to inodes over time.

directory entry to span the deleted entry, an approach taken by similar file systems such as FFS [McKusick et al. 1996]. In ext3cow, the space for unlinked names are not reused, nor are directories compacted. In contrast, FFS reuses space only after all names in a fixed sized chunk are unlinked. Neither approach is particularly attractive. Like both ext3 and FFS, ext3cow will benefit from efficient directory indexing data structures, which is a planned improvement [Ts'o and Tweedie 2002].

5. 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 3). Intuitively, the snapshot number provides access to the file system at the time at which the snapshot was taken. `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 some time $t \geq k$, but remembers the file exists at some time $s \in [j+1,k]$. To restore the file, the user specifies s in the time-shifting interface, `file@s`. The enumeration of versions aids this process; users see all points-in-time at which the file changed using the `ls file@` command and can identify the desired file version.

5.1 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.

5.2 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 time progresses 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.

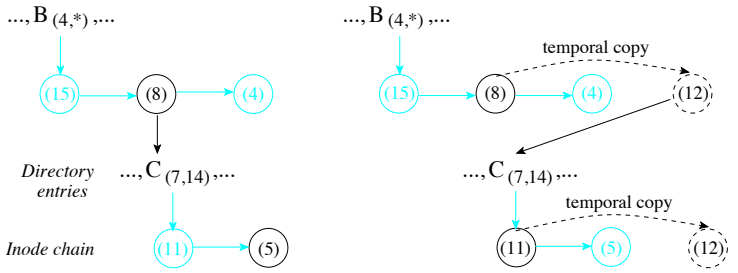
The flexibility of birth/death epoch scoping respects the separation between names and inodes in UNIX-like file systems. Many names may link to a single inode. Also, a different number of names may link to an inode during different epochs. The same name may appear multiple times in the same directory,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

Ext3cow restricts version chains to a constant length through birth/death directory entry scoping. 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 the same data find the new chain. Data blocks may be shared between inodes in the two chains. 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. It will improve the performance of version search from $O(n)$ to $O(1)$ when ext3 adopts extensible hashing for directories.

5.3 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, Figure 5(a) illustrates the wrong version of C being accessed. The access to C



(a) Incorrect scoping.

(b) Correct scoping with temporal vnodes.

Fig. 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.

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 its 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.

6. 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` [McKusick et al. 1996]) 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 ver-

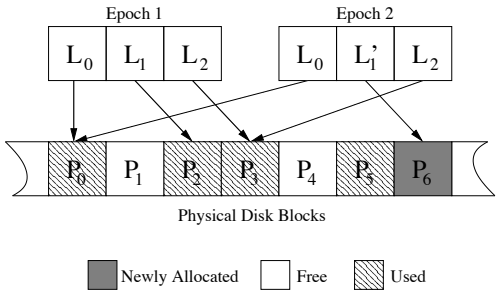


Fig. 6. An example of copy-on-write. The version from epoch 2 updates logical block L_1 into L'_1. Ext3cow allocates a new physical (disk) block P_6 to record the difference. All other blocks are shared.

sions 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 5. 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. 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.

6.1 Memory Management

We isolate the copy-on-write 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` in ext3.

6.2 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. 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 that indirect block, which are also zeroed when creating a copy-on-write version of the indirect block. The bitmap design allows the bitmaps to be updated lazily – only when data are written, not on snapshot.

Because bitmaps borrow space in indirect blocks, the design reduces the maximum file size. However, the loss is less than 10%. Ext3cow represents files up to 15,314,756 blocks in comparison to 16,843,020 blocks in ext3. While larger than 2^{32} bytes, Linux supports 64-bit file offsets. The upcoming adoption of quadruply indirect blocks [Ts'o and Tweedie 2002] will remove practical file size limitations.

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 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.

7. PERFORMANCE EVALUATION

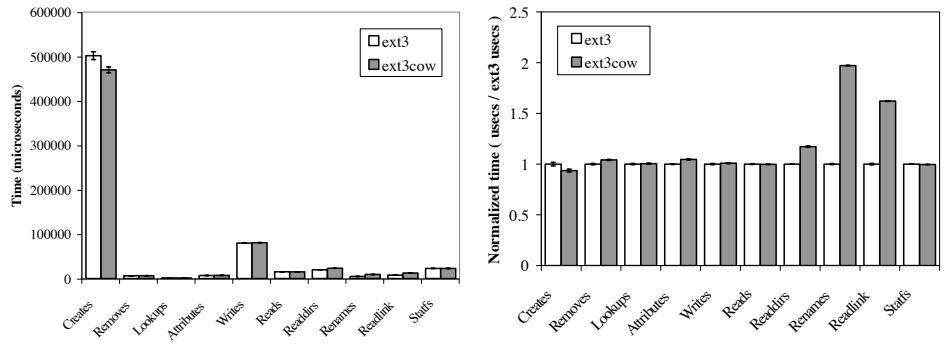
In order to quantify the cost/benefit trade-offs of versioning, we administered a variety of experiments comparing ext3cow to its sister file system – unmodified ext3. Experiments were conducted on an IBM x330 series server, running RedHat Linux 7.3 with the 2.4.19 SMP kernel. The machine is outfitted 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.

7.1 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`,

Operational Test	ext3	ext3cow
Test 1: Creates	501.90 ms	469.94 ms
Test 2: Removes	6.23 ms	6.49 ms
Test 3: Lookups	0.96 ms	0.96 ms
Test 4: Attributes	6.87 ms	7.19 ms
Test 5a: Writes	79.91 ms	80.65 ms
Test 5b: Reads	15.14 ms	15.10 ms
Test 6: Readdirs	19.72 ms	23.12 ms
Test 7: Renames	4.68 ms	9.22 ms
Test 8: Readlink	7.68 ms	12.46 ms
Test 9: Statfs	22.86 ms	22.76 ms

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



(a) Time by benchmark

(b) Time normalized to ext3

Fig. 7. Results from the "basic" tests in the Connectathon benchmark suite. All data are shown with 95% confidence intervals.

(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 20 runs on a newly mounted (cold cache) file system. Ext3cow meets the performance of ext3 in most areas. Table II shows the average cumulative time to perform each test. We present the same data as bar graphs in both absolute time values (Figure 7(a)) and time normalized to the performance of ext3 (Figure 7(b)). Graphs include 95% confidence intervals.

Ext3cow and ext3 perform equally on tests that read inodes and data. Examples include tests 3 (Lookups), 5b (Reads), and 9 (Statfs). On these tests, the file systems execute the same code paths and manipulate the same data structures.

Ext3cow also matches the performance of ext3 when writing and deallocating inodes. Tests 1 (Creates), 2 (Removes), and Test 4 (Attributes) show equivalence. The Attributes and Removes tests navigate a name tree, operating on files rather than inodes directly. The small difference in performance comes from overhead on naming operations. On create, names do not need to be parsed for time scoping.

Benchmark results indicate that ext3cow and ext3 are comparable when writing data

(Test 5a, Writes). In practice, we expect ext3cow to incur a minor penalty on writes. Ext3cow needs to check the copy-on-write bitmap to determine whether a block should be copied the first time a block is written. Subsequent writes to that (dirty) block do not need to check again. The benchmark truncates and rewrites the file anew on each trial, and, therefore, does not exercise this feature.

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 8 (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.

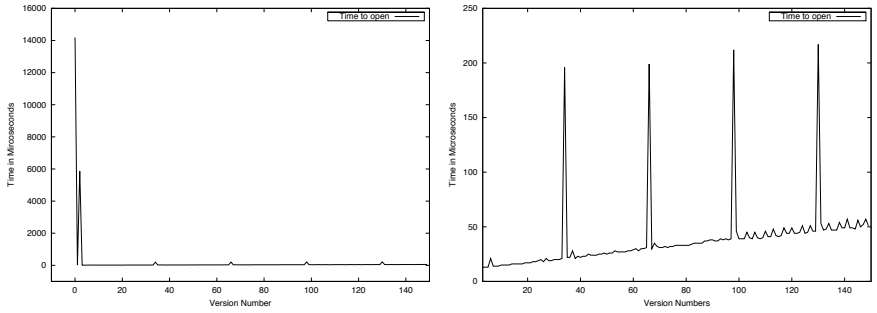
Test 6 (Readdirs) shows the overhead of scoping names in directories. The system does not parse strings or interpret the `@` symbol during this test. Directory names are read and returned to the calling function without interpretation. The overhead comes from directory entry scoping only. Ext3cow examines the birth and death epoch of every record that it reads. This overhead is modest in the benchmark, but might be larger in practice when ext3cow needs to consider more names in a directory – those from previous epochs as well as current epoch.

In total, micro-benchmark results indicate that ext3cow performs comparably to ext3 on data and inode operations. and slightly worse on name operations.

7.1.1 *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 8(a) shows the results 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 8(b) shows a closeup of Figure 8(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.

We attempted many "worst-case" versions of this experiment by artificially aging a file system. In one experiment, we create a block group worth of inodes (>16,000) between successive inodes in a chain. The goal was to subvert and render ineffective ext3's inode clustering. The results of all experiments were indistinguishable from the original experiment, showing an initial penalty for I/O and subsequent access out of caches. Our many attempts to "game" ext3cow were rendered ineffective by the combination of placement policies, read-ahead, and disk (track) caching.

We also conducted experiments that flush the cache between accesses to the k^{th} and $k+1^{th}$ versions. In this experiment, each inode in the chain takes approximately 14 ms to access, because I/O dominates in-memory operations. Performance grows linearly in the number of versions.



(a) Opening Past Inodes (b) Opening Past Inodes (Enlarged)

Fig. 8. The time to open 150 versions of a file.

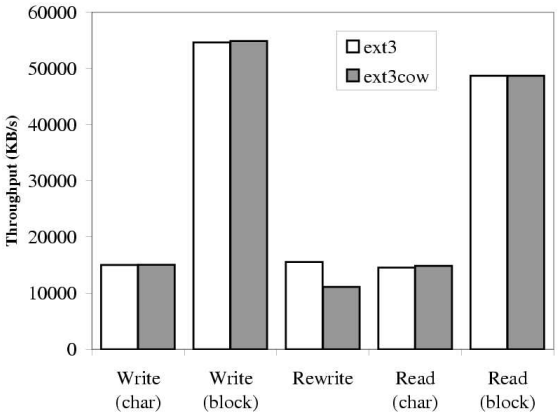


Fig. 9. Results from the Bonnie++ file system benchmark.

7.2 Bonnie++

Bonnie++ is a popular performance benchmark that quantifies five aspects of file system performance based on observed I/O bottlenecks in a Unix-based file system. Bonnie performs a series of test on files of a known size. In our experiments, two, one gigabyte files. This insures I/O requests are not served out of a disk cache, requiring disk access. For each test, the benchmark reports throughput, measured in kilobytes processed per second. The first test measures the rate of sequential character output, while the second test measures sequential block output. The files, in the third test, are then sequentially read and rewritten. Lastly, tests four and five measure the sequential input, by character and by block.

The results of an ext3 and ext3cow Bonnie++ comparison are presented in Figure 9. Ext3cow performs comparably with ext3 in all but the rewrite experiment. This slight performance degradation is due to the copy-on-write bitmap operations that must be performed when rewriting a buffer.

File System	Allocated Blocks	Allocated Inodes	Dir Inodes
ext3	1684696	1243263	15318
ext3cow – none	1684696	1243263	15318
ext3cow – 24 hour	1748126 (+3.8%)	1253642 (+0.1%)	33447 (+218%)
ext3cow – 1 hour	1850189 (+9.8%)	1289513 (+3.7%)	35440 (+231%)
ext3cow – 1 min	2144663 (+27.3%)	1370547 (+10.2%)	64458 (+421%)

Table III. 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.

7.3 Trace-driven Experiments

To examine the effect of snapshot on metadata allocation, we used four months of system call traces from Berkeley [Roselli and Anderson 1996] 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 [Smith and Seltzer 1997]. The traces were played back through two file systems: ext3, 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 file names or directory hierarchy, but do record file creations and directory traversals. For our experiments, we use these operations to infer a directory hierarchy and create file names. Our experiment maintains a consistent mapping of the files we create to the file and device identifiers in the trace. This ensures that operations to the same file in the trace are performed to the same file in our experiment.

Table III displays a 0.1% increase in metadata for 24 hour snapshots and a 10.2% increase for 1 minute snapshots. These results indicate a small initial jump in the amount of metadata to support any amount of versioning, followed by gradual growth as snapshot frequencies increase. These results are consistent with those presented in CVFS [Soules et al. 2003]. Of the 10.2% increase in metadata, 4.7% comes from versioned directory inodes.

Results show the storage cost of indefinite versioning to be quite small for snapshot intervals of an hour or more. Shorter snapshots (1 minute) produce larger overheads, although the storage requirements only increase by 27% over four months. We expect the overhead rate of 27% to be stable over time. The majority of this overhead comes from recent files (older than 1 minute and younger than 1 hour) and, thus, updates to old data do not make up a large portion. Similarly, updates from files older than a day make up only 3.8% of the total 27%. Inode overheads are smaller than block overheads. Directory inode overheads are much greater, ranging to 421%. However, percentage overhead is not the right measure here. The total number of directory inodes is small when compared with all allocated inodes; they make up only 4.7% of all allocated inodes in the one hour snapshot trial and fewer that 3% in all others experiments. Thus, they have a small overall effect on the system.

8. PROJECT STATUS AND FUTURE WORK

Ext3cow is stable and ready for use under the GNU Public License. It is available for download at *http://www.ext3cow.com*. The site, to date, has received thousands of visitors and hundreds of downloads. Beyond its use as an effective file system for end users, ext3cow

is being employed as the platform for additional systems research. The Zap project [Osman et al. 2002] is using ext3cow as the foundation for a virtualization layer that provides groups of processes a consistent view of a system. Ext3cow is also the basis of on-going research on the aging of versioning file systems at U.C. Santa Cruz and practical time-shifting features at U.C. Berkeley. A number of free-lance programmers have asked for support in building network storage devices based on ext3cow.

Our future release plans include a compliance version of ext3cow. In it, we will meet more of the requirements set forth by electronic record management legislation. This includes data encryption, user auditability, and stronger guarantees of the immutability of data in the past. An open-source, regulatory compliant file system will be an essential tool in helping small business control costs while meeting federal mandates on the retention and management of electronic records.

For future research, we are using ext3cow as a platform to solve the problem of secure deletion in versioning systems. The ability to securely delete electronic records is as important as the act of securely maintaining them. Much legislation specifies a period of time for which a company or agency is explicitly liable for their electronic records. For records that fall out of scope, secure deletion removes them forever, making them un-discoverable by subpoena and irrecoverable by malicious attack. Analogous to shredding for paper records, secure deletion is the essential feature in reducing liability for organizations that manage electronic records. The importance of secure deletion is punctuated by the many recent stories in the news about supposedly "deleted" emails that have been recovered to the detriment of their authors.

Presently, permanently deleting data in versioning systems is prohibitively expensive. Even for non-versioning systems, deleting data is inefficient and, thus, rarely performed. Versioning complicate matters, because copy-on-write creates fine-grained data sharing among many versions of a file. Efficient secure deletion in versioning systems is an unsolved problem. The two current methods of secure deletion, secure overwriting [Gutmann 1996] and encryption-key revocation [Boneh and Lipton 1996], suffer from poor performance or fail to meet the requirements of versioning systems. In ext3cow, we are implementing an efficient secure-deletion architecture that deletes large amounts of data based on a small amount of secure overwriting. Our design does not complicate key management, nor does it require any additional secrets to be kept. We recently presented a preliminary design and compared it with existing technologies [Peterson et al. 2004]. We are currently expanding this design and implementing it in ext3cow.

The development of ext3cow has left us with features yet to be implemented. We have referred to several minor enhancements through this paper. These include shell extensions. that will translate a variety of human usable date and time formats to and from the seconds since the epoch format used in time-shifting. They will also support a branching and tagging interface similar to that of CVS [Cederqvist *et al* 2003]. We have also considered taking snapshots on file system subtrees, creating a point-in-time image of a directory and all of its children. However, versioning subtrees is a feature versus efficiency trade off when conducting copy-on-write, because the inode epoch will need to be compared to parent directories recursively to the root of the file system.

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: file system snapshot, per-file versioning, version enumeration, and a continuous-time view of changes to a file system. 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 kernel (virtual file system) or user-space implementations. Given these features, ext3cow supports traditional applications of versioning: easy access to on-line backups; recovery from system tampering; read-only, point-in-time snapshots for data mining; and, file-oriented deletion recovery. However, ext3cow was specifically designed for the management of data in compliance with federal electronic records legislation. As it stands, ext3cow meets the mandated versioning and auditability requirements. In addition, ext3cow's file organization is suitable for the future implementation of secure deletion.

10. ACKNOWLEDGMENTS

The authors would like to thank Avi Rubin and Adam Stubblefield for their contributions to secure deletion. Also, thanks to Jeff Chase, who provided early feedback on this line of inquiry. Lastly, we thank the members of the Hopkins Storage Systems Lab for their support.

REFERENCES

AZAGURY, A., FACTOR, M. E., AND SATRAN, J. 2002. Point-in-time copy: Yesterday, today and tomorrow. In *Proceedings of the Tenth Goddard Conference on Mass Storage Systems and Technologies*. 259–270.

BATTEN, C., BARR, K., SARAF, A., AND TREPETIN, S. 2002. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science. October.

BONEH, D. AND LIPTON, R. 1996. A revocable backup system. In *Proceedings of the 6th USENIX Security Symposium*. 91–96.

BRYANT, R., FORESTER, R., AND HAWKES, J. 2002. Filesystem performance and scalability in Linux 2.4.17. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. 259–274.

CARD, R., TS'O, T. Y., AND TWEEDIE, S. 1994. Design and implementation of the second extended file system. In *Proceedings of the 1994 Amsterdam Linux Conference*.

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

CHERVENAK, A., VELLANKI, V., AND KURMAS, Z. 1998. Protecting file systems: A survey of backup techniques. In *Proceedings of the Joint NASA and IEEE Mass Storage Conference*.

CHUTANI, S., ANDERSON, O. T., KAZAR, M. L., LEVERETT, B. W., MASON, W. A., AND SIDEBOTHAM, R. N. 1992. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*. San Fransisco, CA, USA, 43–60.

CORNELL, B., DINDA, P. A., AND BUSTAMANTE, F. E. 2004. Wayback: A user-level versioning file system for Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 19–28.

CORPORATION, D. E. 1985. *Vax/VMS System Software Handbook*.

EMC Corporation 1998. *EMC TimeFinder Product Description Guide*. EMC Corporation.

GIFFORD, D. K., NEEDHAM, R. M., AND SCHROEDER, M. D. 1988. The Cedar file system. *Communications of the ACM 31*, 3 (March), 288–298.

GREEN, R. J., BAIRD, A. C., AND CHRISTOPHER, J. 1996. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal 8*, 2, 32–45.

GRUNE, D., BERLINER, B., AND POLK, J. 2003. Concurrent versioning system (CVS). http://www.cvshome.org/.

GUTMANN, P. 1996. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*. 77–90.

HAGERTY, J. 2004. Sarbanes-Oxley compliance spending will exceed $5b in 2004. *AMR Research Outlook*.

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

Hitachi, Ltd. 2001. *Hitachi ShadowImage*. Hitachi, Ltd.

HITZ, D., LAU, J., AND MALCOM, M. 1994. File system design for an NFS file server appliance. In *Proceedings of the USENIX San Francisco 1994 Winter Conference*.

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. 1988. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6,* 1 (Feb.), 51 – 81.

HUTCHINSON, N. C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S., AND O'MALLEY, S. 1999. Logical vs. physical file system backup. In *3rd Symposium on Operating System Design and Implementation Proceedings (OSDI)*. 239–250.

JOHNSON, J. E. AND LAING, W. A. 1996. Overview of the Spiralog file system. *Digital Technical Journal 6,* 1, 51–81.

KLEIMAN, S. R. 1986. Vnodes: An architecture for multiple file system in SUN UNIX. In *Proceedings of the 1986 USENIX Summer Technical Conference*. 238–247.

MACDONALD, J. P., HILFINGER, P. N., AND SEMENZATO, L. 1998. PRCS: The project revision control system. In *Proceedings of System Congifuration Management*. Lecture Notes in Computer Science, Vol. 1439, Springer-Verlag.

MCCOY, K. 1990. *VMS File System Internals*. Digital Press.

MCKUSICK, M. K. 2002. Running "fsck" in the background. In *Proceedings of the BSDCon 2002 Conference*. 55–64.

MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley.

MCKUSICK, M. K. AND GANGER, G. 1999. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Freenix Track at the Annual USENIX Technical Conference*. 1–17.

MCKUSICK, M. K., JOY, W. N., LEFFLER, J., AND FABRY, R. S. 1984. A fast file system for UNIX. *ACM Transactions on Computer Systems 2,* 3 (Aug.), 181–197.

MICROSYSTEMS, S. 1989. *NFS: Network file system protocol specification*. Network Working Group, Request for Comments (RFC 1094). Version 2.

MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S. H., AND SMITH, F. D. 1986. Andrew: A distributed personal computing environment. *Communications of the ACM 29,* 3 (Mar.), 184–201.

MOSES, L. 1982. An introductory guide to TOPS-20. Tech. Rep. TM-82-22, USC/Information Sciences Institute.

MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. 2004. A versatile and user-oriented versioning file system. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*. 115–128.

OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. 2002. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*. 361–376.

P. KILLBRIDGE, M. 2003. The cost of HIPAA compliance. *New England Journal of Medicine 348,* 15, 1423–1424.

PATTERSON, H., MANLEY, S., FEDERWISCH, M., HITZ, D., KLEIMAN, S., AND OWARA, S. 2002. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the Conference on File and Storage Technologies (FAST)*. 117–129.

PETERSON, Z. N. J., BURNS, R., AND STUBBLEFIELD, A. 2004. Limiting liability in a federally compliant file system. In *Proceedings of the PORTIA Workshop on Sensitive Data in Medical, Financial, and Content Distribution Systems*.

PRESOTTO, D. 1992. Plan 9. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*. 31–38.

QUINLAN, S. 1991. A cached worm file system. *Software – Practice and Experience 21,* 12 (Dec), 1289–1299.

QUINLAN, S. AND DORWARD, S. 2002. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File And Storage Technologies (FAST)*. 89–101.

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

ROSELLI, D. AND ANDERSON, T. E. 1996. Characteristics of file system workloads. Research report, University of California, Berkeley. June.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10,* 1 (Feb.), 26–52.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementation of the Sun network file system. In *Summer USENIX Conference Proceedings*. 119–130.

SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., AND VEITCH, A. C. 1999. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*. 2–7.

SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. 1999. Deciding when to forget in the Elephant file system. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP)*. 110–123.

SCHROEDER, M. D., GIFFORD, D. K., AND NEEDHAM, R. M. 1985. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP)*. 25–34.

SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. 1993. 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*. 307–326.

SHAPRIO, J. S. AND VANDERBURGH, J. 2002. CPCMS: A configuration management system based on cryptographic names. In *Freenix Track at the Annual the Annual USENIX Technical Conference*. 203–216.

SMITH, K. A. AND SELTZER, M. I. 1997. File system aging – Increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS Conference*. 203–213.

SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. 2003. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. 43–58.

STRUNK, J. D., G. R. GOODSON, M. L. S., SOULES, C. A. N., AND GANGER, G. R. 2000. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*. 165–180.

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

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

TS'O, T. Y. AND TWEEDIE, S. 2002. Planned extensions to the Linux ext2/ext3 filesystem. In *Freenix Track at the Annual USENIX Technical Conference*. 235–243.

ZADOK, E. AND BĂDULESCU, I. 1999. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*. 141–151.

Received Month Year; revised Month Year; accepted Month Year