Causality-Based Versioning

Kiran-Kumar Muniswamy-Reddy and David A. Holland
{kiran, dholland}@eecs.harvard.edu
Harvard School of Engineering and Applied Sciences

Abstract

Versioning file systems provide the ability to recover from a variety of failures, including file corruption, virus and worm infestations, and user mistakes. However, using versions to recover from data-corrupting events requires a human to determine precisely which files and versions to restore. We can create more meaningful versions and enhance the value of those versions by capturing the causal connections among files, facilitating selection and recovery of precisely the right versions after data corrupting events.

We determine when to create new versions of files automatically using the causal relationships among files. The literature on versioning file systems usually examines two extremes of possible version-creation algorithms: open-to-close versioning and versioning on every write. We evaluate causal versions of these two algorithms and introduce two additional causality-based algorithms: Cycle-Avoidance and Graph-Finesse.

We show that capturing and maintaining causal relationships imposes less than 7% overhead on a versioning system, providing benefit at low cost. We then show that Cycle-Avoidance provides more meaningful versions of files created during concurrent program execution, with overhead comparable to open/close versioning. Graph-Finesse provides even greater control, frequently at comparable overhead, but sometimes at unacceptable overhead. Versioning on every write is an interesting extreme case, but is far too costly to be useful in practice.

1 Introduction

Versioning file systems automatically create copies (i.e., versions) of files as they are modified, providing numerous benefits to users and administrators. Users find versions convenient when they inadvertently remove or corrupt a valuable file. Administrators find that versioning systems greatly reduce the rate of requests to restore files from backup. In addition, versioning file systems provide the means to clean up after a data-corrupting intrusion. Unfortunately, versioning alone does not help in identifying the most recent "good" version of a file or how data corruption may have spread from one file to another.

Snapshotting, often implemented using checkpoints, is another approach for versioning that is common for backup systems. Such a system periodically takes a whole or incremental image of the file system and then uses copy-on-write for data modified after the snapshot. Snapshots, similar to versioning file systems, cannot help identify the most recent "good" version of a file. Another drawback of snapshot-based systems is the granularity of recovery: it is not possible to undo changes made between snapshots.

Several new file system designs capture causality relationships among files for a variety of different purposes. For example, Taser [7] captures causality information to address the challenge of identifying files tainted by an intrusion or corrupted by administrative errors. BackTracker [11] captures causality information to analyze intrusions. Provenance-aware storage systems (PASS) capture the provenance or digital history of files to let users answer questions such as, "How do these two files differ?" "What files are derived from this one?" "From what files is this file derived?" "How are these two files related?" [14]. Other systems [19] preserve causal relationships to enhance personal search capabilities.

Combining versioning and the capture of causal relationships introduces functionality not available in existing systems. For example, suppose a system has been compromised by a data-corrupting worm. Upon identifying a tainted file, the causal relationships provide a mechanism to trace backwards to find the last version prior to the corruption and then trace forward to identify *all* the files tainted by that corruption. These two traces precisely identify the appropriate files and versions that need to be restored to recover from the intrusion. Without versioning, an administrator's only recourse is to re-

store the system to a clean snapshot, potentially losing valuable user data. Without causal relationships, the administrator cannot know how the corruption has spread.

Similarly, imagine a scenario where a physics simulator produces results today that differ from those produced yesterday. Here, causal data can reveal the cause of the difference, while versioning data can recover to the earlier (and presumably correct) version.

Conventional versioning file systems [3, 10, 15, 18, 24] typically use one of two techniques to determine when to create new file versions: "open-close" and "version-on-every-write". In the "open-close" approach, versions are defined relative to `open` and `close` events. Typically a new version is created upon the first block update after an `open` and all writes that occur before the final `close` operation appear in that new version. This has the potential to lose valuable information. For example, consider the split-logfile vulnerability in Apache 1.3 [25]. The vulnerability, present in a helper program called `split-logfile`, allows any file in the system with a `.log` file extension to be written. Assume that a database server running on the same machine as the Apache `split-logfile` helper uses a file called `db.log` to store its recovery information. This database server opens the file when it is started and keeps it open. The first time the database server writes to the log file, an "open-close" system will create a new version of it. That version will remain the current version until the database is shut down. Now, suppose an attacker exploits Apache and writes new data in `db.log`. At this point, the log consists of some old "good" log entries and some new "bad" log entries. Even if an administrator finds that `db.log` has been corrupted, the only version available for recovery is the one that existed before the database server wrote anything. If the administrator restores that version, all database operations since the server started will be lost. One might turn to the "version-on-every-write" algorithm, which creates a new version each time data is written to the file; this approach ensures that no data is lost, but it can be expensive in both time and space.

Fortunately, versioning algorithms informed by causality relationships produce versions that facilitate recovery to the pre-tampering state without the overhead of versioning on every write. In the Apache example above, causality-based techniques force a new version of the log file to be started before the attacker's writes are applied. We introduce two such causality techniques: **Cycle-Avoidance** and **Graph-Finesse**. Cycle-Avoidance conservatively declares new versions using knowledge local to the objects being acted upon (i.e, process, files, pipes, etc). **Graph-Finesse** is less conservative, using global knowledge to maintain an in-memory graph of dependencies between objects, declaring a new version whenever adding a dependency edge introduces a cycle. We discuss these algorithms in more detail in Section 4.

As we show in Section 6, any kind of versioning, when coupled with maintenance of causal relationships, provides significant value. In the presence of long-running and/or concurrent execution, Cycle-Avoidance creates the versions necessary to recover from corruption without introducing significant overhead above that of open/close. Graph-Finesse creates slightly fewer versions than Cycle-Avoidance, but it pays significant overhead in workloads that read and write a large number of files. Versioning on every write exhibits sufficiently high overhead that it is impractical.

The contributions of the paper are as follows:

- New functionality arising from the integration of versioning with causal data,
- New causality-based techniques for versioning,
- A prototype embodying versioning and causal relationships, and
- An evaluation of four causality-based versioning algorithms.

The rest of the paper is organized as follows. In Section 2, we introduce the system upon which we build our causality-based versioning system and describe its essential architectural details. In Section 3, we discuss several novel use cases that causality-based versioning enables. In Section 4, we present details of the new versioning algorithms. In Section 5, we discuss the versioning file system implementation. In Section 6, we present evaluation results. In Section 7, we discuss related work. Finally, we conclude in Section 8.

2 Causal Relationships with PASS

We extended PASS [14] (Provenance-aware storage system), the causality-collection system we built, to capture versioning information. We chose PASS as it captured precisely the data that we needed and has a modular architecture that made it easy to add versioning. In this section, we provide a high level overview of the PASS architecture to provide the necessary background to understand our version creation algorithms and implementation.

Figure 1 shows the PASS architecture. Its key components are:

- **Interceptor**: The interceptor intercepts system calls, passing information to the observer, described next. The interceptor is a thin layer that is operating system specific, while the remaining components can be operating system independent.

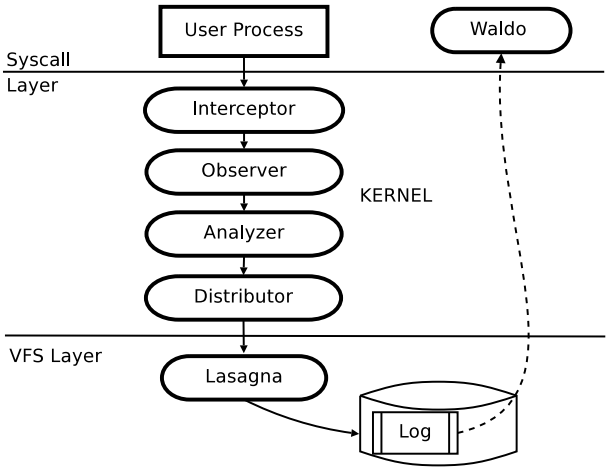

Figure 1: PASS Architecture

- **Observer**: The observer translates system call events to provenance records. For example, when a process P reads a file A, the observer generates a record $P \rightarrow A$, to indicate that the process P depends on the file A. It is precisely these provenance events that capture the causal dependencies in which we are interested.

- **Analyzer**: The analyzer processes the stream of provenance records, making sure that there are no cyclic dependencies among objects. This is where we implement our different versioning algorithms.

- **Distributor**: The distributor maintains provenance for transient objects such as pipes and processes. When these transient objects become part of the ancestry of a regular file on a PASS volume, the distributor creates a virtual object for them on the PASS volume and stores their records to the volume. Creating a virtual object avoids the need for duplicating the provenance of transient objects each time we need to create a causal dependency involving them. Similar to regular files, PASS versions these transient objects when necessary.

- **Lasagna**: Lasagna is the provenance-aware file system that stores provenance records along with the data. Internally, lasagna writes the provenance into a log.

- **Waldo**: Waldo is a user-level daemon that reads provenance records from the log and stores them in a database. After a data corruption, PASS's recovery tools consult this database to determine causal relationships.

The observer and analyzer are central to causality-based versioning and are discussed further in Section 4.

PASS maintains provenance, and therefore causal relationships, for both persistent and transient objects. A relationship between two files (persistent objects) is therefore expressed indirectly as two relationships, each between a file and a process. For example, when a process issues a `read` system call, PASS creates a record stating that the process causally depends upon the file being read. When that process then issues a `write` system call, PASS creates a record stating that the written file depends upon the process that wrote it. The causal relationships described by these records are the heart of the data we use, both to instantiate file versions and also to choose files to restore after a corruption.

3 Use Cases

In this section, we discuss use cases that demonstrate the novel functionality enabled by causality-based versioning.

3.1 Intrusion Recovery

Recently, one of the authors upgraded the software on his system. The upgraded packages included `coreutils`, the package that contains `ls`. This author completed the upgrade and continued working for the rest of the evening. However, when he came back the next morning, `ls` emitted the message:

```
/bin/ls: unrecognized prefix: do
/bin/ls: unparsable value for
         LS_COLORS environment
         variable
```

The author then searched the web to learn that the behavior might be the result of an intrusion. He promptly installed `chkrootkit` and `rkhunter`, two popular programs to verify that a system has been hacked. However, both the programs failed to locate any known rootkits. At this point, it was unclear if the aberrant behavior of `ls` was due to the update to `coreutils` the evening before or an intrusion.

Had he been running PASS, the author could have followed the chain of provenance dependencies of the file `/etc/DIR_COLORS`. (`LS_COLORS` is derived from `DIR_COLORS`). If the provenance chain of `DIR_COLORS` indicated that it was modified by the package manager or a legitimate system utility, then the system had not been hacked; otherwise, the system had been hacked. In the event the system had been hacked, there would be only one option: *wipe the system and reinstall.* This is obviously undesirable, especially since the author had recently completed a re-install in order to

upgrade. Faced with this situation, the author longed for a versioning file system coupled with PASS that would permit him to selectively roll back the affected files to the version just before they were corrupted. Had that been an option, he could have continued using his system without a full re-install.

3.2 Reproducing Research Results

Systems that collect provenance frequently do so to facilitate the reproduction of scientific results [22]. Consider the common scenario where a scientist collects data from some device (e.g., a telescope), transforms it through many intermediate stages and produces a final output file. Suppose he finds, a few months after publishing the data, that one of the programs in the intermediate stages had a bug. The scientist has no option but to begin anew with the raw data and then re-run all the experiments that he thinks may have been affected by the bug.

If, however, he has complete provenance of his data, he can identify precisely which data sets were affected by the corrupt program. Hence, he need only re-run those experiments from the point at which the corruption occurred. Further, if the raw data is unavailable (frequently, raw data is archived and removed from data processing systems once it has undergone its initial pre-processing), he can use a versioning system to recover the missing raw data to re-run the experiments. This method obviates the need to retrieve the raw data from archives or a central repository.

3.3 System Configuration Management

Software configuration management is extremely hard [26]. New software installation can (and regularly does) break existing software, because packages interact with each other through various agents: libraries, registries, configuration files, and even environment variables.

Provenance systems can help alleviate some of the problems of configuration management by helping users recover from a corrupt configuration. One of our authors recently installed a new music player on his system. The music player, in turn, depended on a number of libraries that needed to be updated or downloaded. After the install, the music player worked well, but much to the annoyance of the author, his movie player ceased to work. The author guessed that is was probably because of the updates to the libraries. The author tried to use the package manager to revert the system to the state that existed prior to the music player install. Using the system package manager to remove the music player did not help, because as far as the package manager was concerned, the library updates were independent of the music player install. The author could not manually undo the library updates as he did not know the list of libraries that were installed. A record of the causal relationships between the libraries, the music player, and the movie player would have helped the author identify which of the libraries were common to the music player and the movie player and hence would have helped to point out (or narrow down) the offending library. Such causal data coupled with a versioning file system provides exactly the information needed to permit the user to revert all the modified files to a state prior to that of the music player install. Since the versioning system even restores the package manager database to its prior state, it preserves the consistency of the system.

The problems outlined in this use case arise mainly because package management dependencies are generated manually and are brittle in nature. Alternatively, one could use causal data recorded by PASS to gather the true dependencies of a package that, in turn, can help perform better roll backs after installation.

3.4 Database Recovery

Traditional databases are designed to recover from software and hardware crashes. However, those mechanisms are not sufficient to recover from a human error or a compromise due to the time gap between the event occurrence and the detection. In such cases, recovery involves a manual sanitizing of the database. Causality-based versioning can help reduce the amount of effort and the downtime of the service as we show in the following example. For simplicity, we assume that the database is not running transactionally.

A faulty client can corrupt a database by either adding incorrect entries, removing valid entries, or updating existing entries incorrectly. Once the faulty client is detected, one can use the causal data collected by PASS with the versioning data to recover the database. Recovery is simple in the case where the last database update is by the faulty client. In this case, recovery simply entails reverting the database to a version before the client updated it. In the case where legitimate actions are interleaved with the actions of the faulty client, automatic recovery is hard as both legitimate and faulty updates are coalesced in main memory and then written to disk. In this scenario, one can use causal information to recover the database to a version before the faulty client's modifications and a version after the faulty client's modifications and then compute a difference of the data dump between the two versions. The difference in the data dump will contain both legitimate and illegitimate data that needs to be sanitized, but the number of rows requiring manual checking is much smaller than the entire database.

3.5 Intellectual Property Compliance

Causality-based versioning can also help verify intellectual property (IP) compliance and enable removal of IP violations. For example, companies that use and develop both proprietary software and open source software routinely require pre-release checks to make sure the proprietary software has not been tainted by open source software and vice-versa. In most cases, this is a tedious, manual process. One can use causal relationships to identify paths between source files with different licensing models. When coupled with a versioning file system, the system supports rigorous analysis of such license pollution and potentially explicit means of reverting to untainted states.

4 Versioning Algorithms

As described in Section 2, in the PASS architecture the *observer* generates causal relationship data and the *analyzer* prunes these relationships, removing duplicates and cycles. Programs generally perform I/O in relatively small blocks (e.g., 4 KB), issuing multiple reads and writes when manipulating large files. Each `read` or `write` call causes the observer to emit a new record, most of which are identical. The analyzer removes these duplicates. Meanwhile, cycles can occur when multiple processes are concurrently reading and writing the same files [2]. Cycles in causality are nonsensical and must be avoided. In the PASS system, the analyzer prevents cycles by forcing new versions of objects to be created. It does this by choosing when to *freeze* them; that is, when to declare the current version "finished" and begin a new version. Transient objects (processes and pipes) can also be frozen to break cycles. To experiment with causality-based versioning, we installed the versioning algorithms in the analyzer. In this section, we describe the versioning algorithms we used, referencing Table 1 as an example sequence of events.

4.1 Traditional Algorithms

In the open-close (OC) algorithm, the last close of a file (that is, when no more processes have the file open) triggers a freeze operation. The next open and write triggers the start of a new version. This algorithm does not preserve causality; some sequences of events (including the example in Table 1) produce cycles. Figure 2 illustrates how this happens.

The version-on-every-write (ALL) algorithm creates a new version on every write. This avoids any violations of causality but potentially creates a large number of versions. In this sense, it is the most conservative of the algorithms we consider. The code for this algorithm is quite simple; because each write results in a new version of a file and each read results in a new version of a process, each record refers to a distinct version of something. Thus, there is no need to check for either duplicates or cycles.

Step	P	Q
1	read A	
2		read B
3	write B	
4		write A
5	read A	
6		read B

Table 1: Example scenario to illustrate the versioning algorithms. Each `read` and `write` operation is enclosed by an `open` and `close`. All objects are initially at version one.

4.2 Cycle-Avoidance

The Cycle-Avoidance (CA) algorithm, as its name suggests, preserves causality by avoiding cycles. For each object, the analyzer maintains a unique object ID (assigned at object creation), a version number (incremented on each freeze), and an *ancestor table*. The ancestor table records the object ID and version number of all the immediate ancestors of the object. When CA receives a record of the form $A_i \rightarrow B_j$, it stores B_j in the ancestor table of A. CA creates a new version of an object whenever it adds a *new* ancestor, where different versions are considered distinct, to the object's ancestor table. Doing so guarantees that no cycles will be created. CA differs from version-on-every-write, because not all writes add new ancestors.

When the analyzer receives a record of the form $A_i \rightarrow B_j$, it examines the ancestor table of A for B_k, that is, some version k of object B, and uses the following rules to perform both duplicate detection and cycle handling.

- **Rule CA.1**: If no B_k exists in the ancestor table of A, then B is a new ancestor for A. Issue a freeze operation on A to create a new version and add B_j to the ancestor table of A.
- **Rule CA.2**: If B_k exists and $j = k$, then the causality record $A_i \rightarrow B_j$ is a duplicate and we discard the record.
- **Rule CA.3**: If B_k exists and $j < k$, the new record refers to a version older than the most recent one recorded in A's ancestor table, and the existing causality relationship $A_i \rightarrow B_k$ subsumes any causal relationships of B_j. Hence, the causality

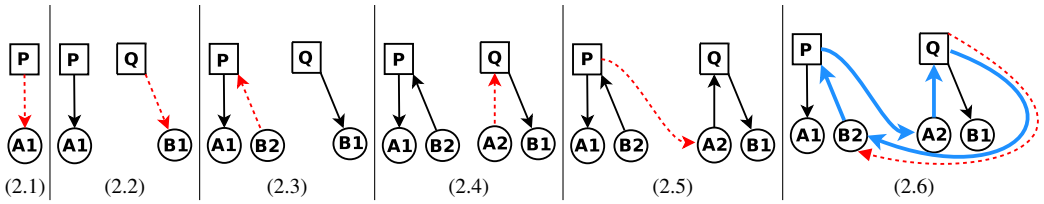

Figure 2: Illustration of the open-close algorithm for the sequence in Table 1. The arrows represent causality and point opposite to data flow. In (2.3), a new version of B is created as it is the first write since the last close. In (2.4), a new version of A is created for the same reason. A cycle $A_2 \rightarrow Q \rightarrow B_2 \rightarrow P \rightarrow A_2$ (thick lines) results on the last read as shown in (2.6).

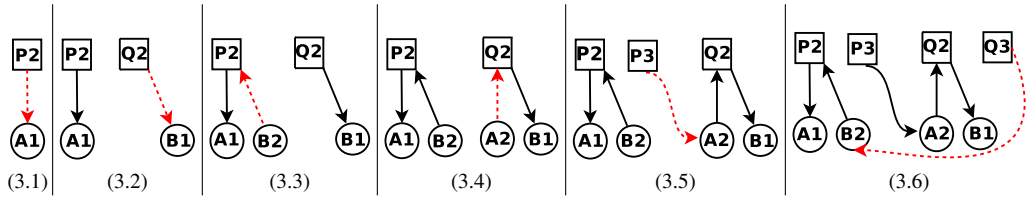

Figure 3: Illustration of the Cycle-Avoidance algorithm for the sequence in Table 1. In (3.1), a new version of P is created by the rule CA.1. In (3.2), a new version of Q is created by the rule CA.1. In (3.3), a new version of B is created by the rule CA.1. In (3.4), a new version of A is created by the rule CA.1. In (3.5), a new version of P is created by the rule CA.4. In (3.6), a new version of Q is created by the rule CA.4. The end result is that there are no cycles.

record $A_i \rightarrow B_j$ is a duplicate and we discard the record.

- **Rule CA.4**: If B_k exists and $j > k$, B_j is a newer version than the B_k in A's ancestor table. Thus, B_j depends on some objects on which B_k did not depend. Therefore, we issue a freeze on A to create a new version and update the ancestor table of A to name B_j instead of B_k.

Figure 3 illustrates the behaviour of the Cycle-Avoidance algorithm for the example sequence in Table 1.

4.2.1 Self-Cycles

Self-cycles arise when a process is both reading and writing the same file. Some programs, such as the GNU linker, generate self-cycles as they repeatedly read from and write to their output files. The Cycle-Avoidance algorithm as described can create a large number of unnecessary versions in this situation. To avoid this, we track each object's *last ancestor*. When the analyzer receives a record of the form $A_i \rightarrow B_j$, it makes the following check:

- **Rule CA.self-cycle**: If the last ancestor of B_j is A_i, the new record creates a self-cycle; discard the record.

The above rule tells us that the last version change of B occurred *because* of the current version of A. In that case, the data being fed to A originated from A itself, and we have a self-cycle. Records representing self-cycles do not add information and can be dropped immediately.

4.3 Graph-Finesse

As described above, Cycle-Avoidance decides when to create new versions using local knowledge about the object to which a dependency refers. In contrast, Graph-Finesse (GF) uses global knowledge to make its decisions. It maintains a global directed graph of the causal dependency relationships between objects. The GF algorithm checks each new record against the graph and forces the creation of a new version of a single file if and only if adding the record would otherwise create a cycle. The name arises from the fact that it picks out a comparatively small number of new versions to create while still preserving causality.

Given a record $A_i \rightarrow B_j$, GF uses the following rules:

- **Rule GF.dup**: Check if $A_i \rightarrow B_j$ already exists in the causal-dependency graph. If so, the record is a duplicate; discard it.

- **Rule GF.detect**: Check if $B_j \rightarrow^* A_i$, that is, if a path of zero or more steps exists linking B_j to A_i. If so, then $A_i \rightarrow B_j \rightarrow^* A_i$ forms a cycle. Freeze A, creating A_{i+1}, change the record to $A_{i+1} \rightarrow B_j$, and add this information to the graph. There will now be no cycle; because A_{i+1} is new, it cannot be an ancestor of B_j.

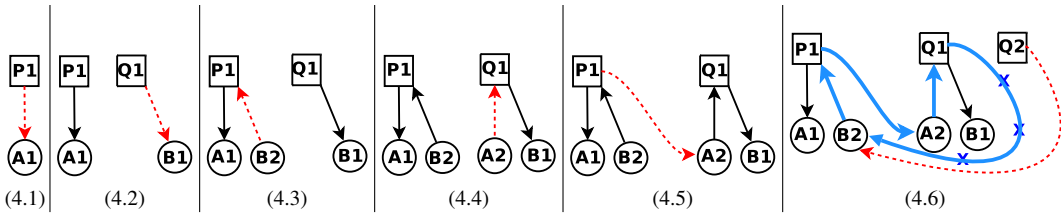

Figure 4: Illustration of the Graph-Finesse algorithm for the sequence in Table 1. In (4.3) and (4.4), new versions of B and A are created by the open-close mechanism. In (4.6), a path exists from B_2 to Q_1 (thick lines), so Q_2 is created by rule GF.detect and no cycle is formed. As we can see, Graph-Finesse creates fewer versions than Cycle-Avoidance.

- **Rule GF.default**: Otherwise, add $A_i \rightarrow B_j$ to the graph.

By design, GF also subsumes open-close versioning and includes the freeze-on-last-close behavior as described in Section 4.1.

To keep the graph from growing without bound, it is important to *prune* it. Any node in the graph (representing some version of some object) may be pruned if that node is *frozen*, that is, any future write to the object will create a new version or be to some already existing newer version, and all the ancestors of the node are frozen. (If an ancestor is unfrozen, writing to it may cause a cycle.)

It is therefore possible to bound the size (or diameter or other measure) of the graph at the cost of creating extra versions, by freezing unfrozen objects that would otherwise be pruned. We have considered this but not implemented it, because there seems to be little need for it in practice.

Graph-Finesse can consume more CPU for some workloads as it has to traverse the graph on every record addition. Our evaluation confirms this. The memory consumption of Graph-Finesse, however, is comparable to the other algorithms. Graph-Finesse can also use the self-cycle logic described in Section 4.2.1. Figure 4 illustrates the behaviour of the Graph-Finesse algorithm for the example workload.

4.4 Discussion

We now discuss how pruning entries in CA and GF does not affect the use cases discussed in section 3. The OC, CA, and GF algorithms record the same versioning data for the first three use cases as they involve an application making a one time modification to all the data files involved. Such changes generate the same causal information and data versions for the three algorithms. The ALL algorithm generates the same causal information and versioning data as the other algorithms with the difference being that the data is spread over more versions. For the remaining two use cases, the causal algorithms record different causal information and versioning data. We explain how they differ in the database recovery (3.4) use case.

The database server (server) opens the database at startup, writes to it in response to client requests, and closes the file at shutdown. In OC, the first time the server writes to the database, it creates a new version. All subsequent database modifications are part of this new version and old data is not copied before applying these modifications, because the file is still open. Thus, restoring the old version loses any legitimate modifications between the first write and the faulty writes. Clearly, OC versioning is not sufficient for the database recovery use case. ALL has sufficient information as it recorded all causal information and versioned on every database modification. However, versioning the database this frequently is potentially very inefficient.

In CA, reception of a client request produces a new version of the server process. This, in turn, triggers a new version of the database, when the server modifies the database. Multiple modifications resulting from a single client request do not create multiple versions, because the server's version does not change. However, when a new (faulty) request arrives, the server's version increments as does the database's version. Interleaved requests from multiple clients will generate many versions. Thus, in a single client case, CA behaves like OC and when clients interleave, it behaves like ALL. Hence, there is sufficient information to undo a faulty client's updates. The GF algorithm behaves in a manner similar to CA.

5 Implementation

In this section, we describe our versioning design choices, our implementation, and the limitations of our system. Our versioning design was influenced by the comprehensive versioning file system (CVFS) [24], which explored metadata efficiency in versioning file systems. CVFS showed that logging all the modifications of a file to a journal is more efficient than creating a new inode for each version. Hence, we use a redo log for storing versioning data for files. Although CVFS

uses multiversion B-trees to handle versions of directories, we store directory metadata in an undo log, as this is much simpler to implement. We implement our versioning system by modifying Lasagna, the PASS storage engine. Lasagna is a stackable file system, which used eCryptfs [8] as its starting code base.

5.1 File Versioning

For each file `foo`, we maintain a version file `v;12345`, where `12345` is the inode number of `foo`. The version file is a log where we record old data before updating the primary file. Inode numbers are never reused, because we never delete files. For locality, we keep the version files and the files they describe in the same directory. Users cannot access the version file directly as we filter out the version file names in `readdir` and `lookup`.

The version file consists of the following three types of log records. The *version* record marks the start of data records for a version. It contains the version number and the metadata attributes of the version such as the file size, uid, gid, etc. The *page* record holds old data being overwritten. This contains the data and the page number in the file from which the data came. Finally, the *beginptr* record is the last record of a version; it records the location of the corresponding *version* record to allow scanning backwards.

Each version begins with a *version* record, ends with a *beginptr*, and has some number of intervening *page* records. We write a *beginptr* record to the version file when a freeze request is issued on the file. We write a *version* record on the first `write` call on the file after a freeze.

When a program issues a `write` call on a file, we read the pages that the `write` call overwrites and write a *page* record for each. When a file is truncated, we log all discarded pages. We record each page only once per version. For example, if the file system receives two 4K writes at offset 0, we log data only for the first (assuming the version does not change between the writes). On an `unlink`, we rename the target file to `v;12345;deleted` where `12345` is the inode number. A native file system could remove the file blocks from the primary file and append them to the version file. Lasagna, however, is a stackable file system and does not control the file layout of the underlying file system. Instead, we `rename` the file on the last `unlink`. This is more efficient than copying blocks from the primary file to the version file, especially for large files.

5.2 Directory Versioning

For each directory, we maintain (within the directory) a version file named by inode number, as we do for files. The directory version file has *version* and *beginptr* log records as we do for a regular file. It also has three other log record types: The *add entry* record represents an addition to a directory via `create`, `link`, `mkdir`, or `symlink`. This record contains the inode and version of the directory to which we are adding, and the name, inode, and version of the entry, which can be a file or a directory. The *del entry* record represents a removal from a directory by `unlink` or `rmdir`. This contains the same data as the *add entry* record. The *rename entry* records a directory entry being moved from one directory to another. Where appropriate, this is written to the version logs of both directories involved in the rename. This record contains the inode and version of the new directory, the old directory, the old file, and the overwritten target file (if it existed), and the old and new file names.

Because version logs are never deleted and files are renamed on deletion, directories are never truly empty, so our versioning file system cannot perform `rmdir` operations. Instead, when a directory is removed, we check to see if all the files in the directory are either *v;inode* or `v;inode;deleted` files, i.e., all files are either version files or deleted files. If so, the directory is "virtually" empty, and we can move the directory out of the way using a `;deleted` suffix.

5.3 Accessing Previous Versions

We provide an `ioctl` that is used to access old versions of a file. The `ioctl` takes as input, a name, a version, and a file descriptor and recovers the old version into the file descriptor. Internally, we perform recovery by scanning backward in the version file until we find the desired *version* record. Once this has been found, we scan forward in the version file writing the data pages of the file version to the user supplied file descriptor. We also update the attributes of the file descriptor based on the values recorded in the *version* record. Hence, previous version access is a *redo* operation. Directory operations, on the other hand, are undone depending on the contents of the version log records.

5.4 Limitations

The causal data that we capture is an approximation and can lead to false positives or negatives while performing analysis for recovery. For example, `/etc/shadow` will be in the history of every process and file, as `login` reads it while authenticating users. Hence even legitimate users that log in after an attack can appear to be causally related to the attack. The general approach to deal with this has been to white-list some of these files, i.e., ignore the causal information on some files while performing analysis. Further, contextual policies to ig-

nore some of the causal information can help improve the results of the analysis. For example, to construct the list of files needed to migrate an application, we need to use all the causal information as we do not want the application to fail on restart. However, while analyzing causal data during intrusion recovery, we need consider only those files that have been written by illegal processes.

As with all provenance systems, PASS cannot capture causal dependencies external to the computer. For example, when a user prints a file and then makes some notes based on what she read, PASS cannot capture the dependency between the notes to the source file. PASS does, however, allow users and applications to annotate user-knowledge or application specific provenance to the provenance collected by PASS (with the obvious limitation that the user or application needs to take the correct action).

Attackers can perform a denial of service attack by repeatedly overwriting files, filling the disk with versioning information. While this is not different from an attacker filling the disk with regular files, we can do better than regular file systems by using the causal information to detect anomalous behaviour and prevent it [12, 23].

Because our implementation is a stackable file system, it is vulnerable to tampering by means of unmounting it and inspecting or altering the underlying state. This could be prevented by using cryptography or by having a non-stackable implementation and using a `securelevel`-type scheme to protect raw disk devices.

Finally, we are vulnerable to an intruder changing the kernel. Once the intruder has access to the kernel, she can change the causal information and the versioning information, thus making accurate recovery impossible. Secure Disk Systems [27], where an intruder cannot modify the causal or versioning data once it has been written to disk, helps solve the problem partially, by allowing users to recover data up to the point the system was subverted. This is better than a clean system install. Causal versioning is still useful for recovery in the cases where attackers do not care to cover up their tracks, such as when they set up a bot on a machine and abandon the system after a few days.

6 Evaluation

The goal of our evaluation is twofold. First, to quantify the overheads introduced by the different versioning systems. Second, to evaluate the efficacy and performance of the different algorithms during recovery of files. We address these goals as follows: First, we discuss the evaluation platform and the configurations we used for evaluation. In Section 6.1, we discuss the performance overheads for four benchmarks representative of a broad range of workloads. In Section 6.2, we discuss how the versioning algorithms perform during recovery.

We ran all the benchmarks on a 3GHz Pentium 4 machine with 512MB of RAM. The machine had a 80GB 7200 RPM Western Digital Caviar WD800JB hard drive that was used to store all file system data and metadata, including causality. The machine was running Fedora Core 5 with a PASS kernel based on Linux 2.6.23.17 and Lasagna was stacked on Ext2. We recorded elapsed, system, and user times, and the amount of disk space utilized for all tests. We also recorded the wait times for all tests; wait time is mostly I/O time, but other factors such as scheduling time can also affect it. We compute wait time as the difference between the elapsed time and system+user times. We do not discuss the user time as it is not affected by the modifications in the kernel. We also ran the same benchmarks on Ext2 using those results as a baseline. In order to separate the overhead due to versioning from the overhead due to causality, we also ran all experiments using versioning without enabling causality collection. We used the open-close algorithm for the latter experiments. We ran each experiment at least 5 times. In all cases, the standard deviations were less than 5%.

We evaluate the system under the following configurations:
VER: open-close versioning with no causal data
OC: open-close versioning with causal data
CA: Cycle-Avoidance versioning with causal data
GF: Graph-Finesse versioning with causal data
ALL: Version-on-every write with causal data

6.1 Performance Overhead Results

We ran the following four workloads to evaluate the versioning algorithms. 1. Linux compile, in which we unpack and build Linux kernel version 2.6.19.1. This benchmark represents a CPU-intensive workload. 2. Postmark, that simulates the operation of an email server. This represents an I/O-intensive workload. We ran 1,500 transactions with file sizes ranging from 4 KB to 1 MB, with 10 subdirectories and 1500 files. 3. Mercurial activity benchmark, where we start with a vanilla Linux 2.6.19.1 kernel and apply, as patches, each of the changes that we committed to our own Mercurial-managed source tree. This benchmark evaluates the overhead a user experiences in a normal development scenario, where the user works on a small subset of the files over a period of time; 4. A biological *blast* [1] workload that is representative of a scientific workload. The workload finds protein sequences that are closely related in two different species. The workload formats two input data files with a tool called *formatdb*, processes the two files with *blast*, and then massages the output data with a

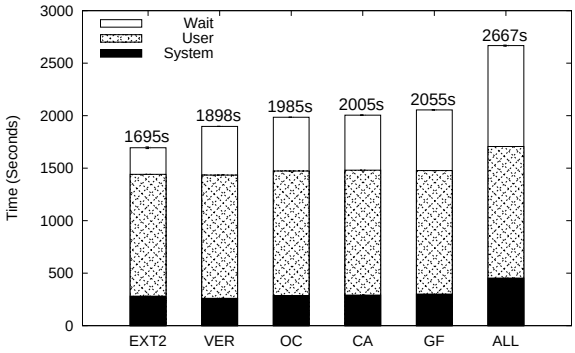

Figure 5: Linux compile elapsed time results.

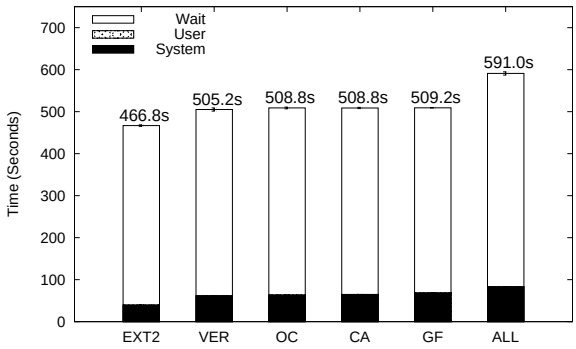

Figure 6: Postmark elapsed time results.

series of *perl* scripts.

	Causal Data	Version Space
VER	-	37.6MB (2.9%)
OC	154.5MB (12.0%)	49.0MB (3.8%)
CA	172.3MB (13.4%)	54.8MB (4.3%)
GF	154.5MB (12.0%)	49.0MB (3.8%)
ALL	462.6MB (35.9%)	1.1GB (85.7%)

Table 2: Linux compile Space overheads. All the overheads shown are computed as a percentage of the data in vanilla Ext2 (1.26GB).

Linux Compile Benchmark Results Figure 5 shows the elapsed time results for Linux compile and Table 2 shows the space overhead. Plain versioning (VER) adds 11.9% to the elapsed time and 2.9% to the space. The increase in elapsed time is mostly due to the additional writes performed to store versions, but a small portion is due to the fact that we use a stackable file system. For the OC, CA, and GF algorithms, the overheads increase moderately over VER to 17.1%, 18.3% and 21.3% respectively. This increase is due to the extra writes issued to record causal data. For this benchmark, CA and GF, the causality based algorithms, perform comparably to OC in terms of elapsed time and version space. ALL, as we expect, has the worst elapsed time performance with 57.4% overhead. The ALL overhead is a result of the enormous number of versions being created and the quantity of data necessary to do so. The system time also increases significantly for ALL due to the distributor having to cache large amounts of causal data.

Postmark Benchmark Results Figure 6 shows the elapsed time results for Postmark and Table 3 shows the space overheads. The overheads follow a pattern similar to the overheads for the Linux compile benchmark. VER has the lowest overhead at 8.2%. VER's overhead is due to the extra writes to record version data and the double buffering in Lasagna (stackable file systems cache both their data pages and lower file system data pages). The overheads increase marginally for OC, CA, and GF to 9%, 9%, and 9.1% respectively. The increase is marginal as causal information recorded is minimal and the version data also increases minimally from VER to OC, CA, and GF. Once again, ALL, with a 26.6% overhead exhibits the greatest overhead as expected.

	Causal Data	Version Space
VER	-	1.28GB
OC	1.8MB (0.14%)	1.28GB
CA	1.2MB (0.09%)	1.28GB
GF	1.9MB (0.15%)	1.28GB
ALL	61.2MB (4.74%)	1.38GB

Table 3: Postmark space overheads. Causal data overheads are computed as a percentage of the data written in Ext2 (1.26GB). Postmark deletes all files it creates at the end of the benchmark. In versioning systems, however, no file is deleted and all unlinked files are retained as is. The version space column shows the amount of space retained at the end of each algorithm.

Mercurial Activity Results Figure 7 shows the elapsed time results for the Mercurial activity benchmark and Table 4 shows the space overhead. The performance overheads follow the pattern we have seen so far. However, surprisingly, GF performs worse than even ALL for this benchmark. VER, CA, GF, and ALL have overheads of 25.9%, 28.8%, 27.9%, 89.6%, and 61.3% respectively. The performance overheads of GF is a result of a very large patch combined with the way the program `patch` functions. `patch` works by first reading the patch file and the file to patch, then merges the two files into a temporary file, and finally renames the tem-

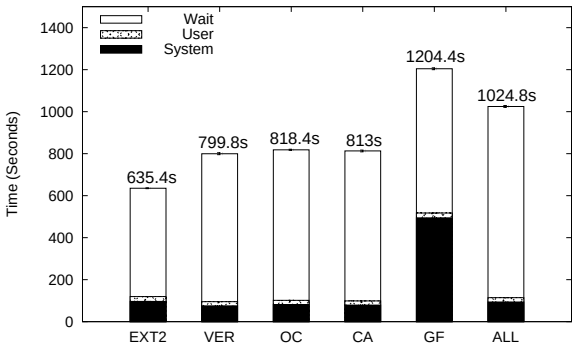

Figure 7: Mercurial activity elapsed time results.

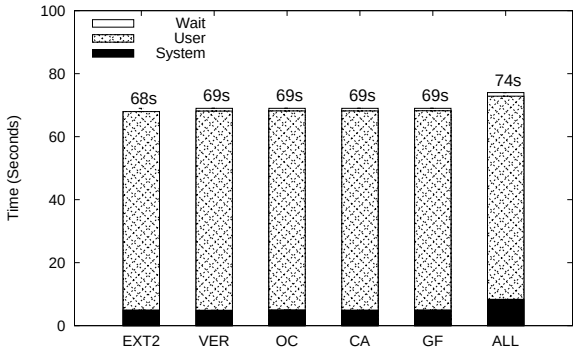

Figure 8: Blast elapsed time results.

	Causal Data	Version Space
VER	-	228.1MB (26.6%)
OC	38.3MB (4.5%)	233.4MB (27.2%)
CA	28.3MB (3.3%)	230.6MB (26.9%)
GF	30.3MB (4.7%)	233.4MB (27.2%)
ALL	77.8MB (9.1%)	383.3MB (44.6%)

Table 4: Mercurial activity space overheads. All the overheads shown are computed as a percentage of the data in Ext2 (859MB).

	Causal Data	Version Space
VER	-	40KB (0.7%)
OC	172KB (2.9%)	40KB (0.7%)
CA	176KB (3.1%)	40KB (0.7%)
GF	172KB (2.9%)	36KB (0.6%)
ALL	3.7MB (65.4%)	14.4MB (257.4%)

Table 5: Blast workload space overheads. All the overheads shown are computed as a percentage of the data in Ext2 (5.8MB).

porary file to the file specified in the patch. At one point during development, we moved from a Linux 2.6.19.1 kernel to a Linux 2.6.23.17 kernel. This resulted in a large patch touching all the source files in the repository. This forced a single instance of `patch` to read and write all of the 20,000 files in the Linux source tree. Every time `patch` writes to a new file, GF verifies that the file does not form a causality-violating cycle with the files that `patch` previously read. This results in the heavy system time overheads. This problem could be alleviated by having `patch` spawn multiple processes each of which merges a unique subset of the files specified in the patch file.

Another anomaly is that CA generates less causal data than OC. The explanation is that this workload generates a rename for every file to be patched. OC issues a freeze on the directory every time a directory is modified. CA, however, uses the causal history to determine that the same process is modifying the directory and eliminates duplicate entries. This results in CA consuming the least amount of both causal and version data.

Blast Workload Results Figure 8 shows the elapsed time results for the blast workload and Table 5 shows the space overhead. The overheads for this workload follow the pattern seen in the previous workloads. The time overhead is 1.4% for the VER, OC, CA, and GF configurations. The causal data overhead is less than 3.1% and the version data overhead is less than 1% for VER, OC, CA, and GF configurations. The workload is CPU intensive and processes a small number of large files, resulting in the minimal overheads for VER, OC, CA, and GF. For ALL, the elapsed time overhead is 8.8% and the space overhead is 65.4% on causal data and 257.4% on version data. The version data overheads of ALL is due to the behaviour of `formatdb` and `blast`. They write data in chunks smaller than a page, which versions the same page multiple times.

6.2 Recovery Benchmark Results

The goal of this subsection is to answer the following questions: First, in scenarios where open-close is sufficient (such as the first three use cases in Section 3), do the (unnecessary) causality-based algorithms impose additional recovery overhead? Second, in scenarios where causality does matter (the last two use cases in Section 3), how do the algorithms compare in recovery time and data loss?

To answer the first question, we wrote a program that simulates the behaviour of a worm. Worms typically overwrite one or more files/executables (for example, common executables like `ls`) and install some new

	Causal Data	Version data	Bytes Read			Recovery Time		
			11^{th}	7^{th}	3^{rd}	11^{th}	7^{th}	3^{rd}
OC	60KB	12KB	-	-	-	-	-	-
CA	176KB	470.5MB	39.15MB	39.16MB	39.17MB	23s	25.2s	27.4s
GF	184KB	470.5MB	39.15MB	39.16MB	39.15MB	23s	24.6s	25.8s
ALL	76.9MB	1.97GB	45.24MB	53.99MB	62.76	214.2s	452.8s	689s

Table 7: Results for recovering from the Apache simulator. All algorithms recover the same amount of data (40MB), but read in different amounts of data to perform the recovery.

	Causal Data	Version Space	Recovery Time	Bytes Read
OC	21.2MB	151.6MB	173.4s	179.1MB
CA	12.7MB	150.4MB	163.2s	163.9MB
GF	21.1MB	151.9MB	173.2s	179.1MB
ALL	52.8MB	249.3MB	191.2s	182.9MB

Table 6: Results for recovering from a worm attack. All algorithms recover the same amount of data (161.68MB), but read different amounts of data to perform the recovery.

files/programs (irchat servers being a popular choice). Our worm-simulator functions in a similar manner. The program traverses a copy of the Linux-2.6.19.1 source tree, overwrites some files and creates new "bad" files. All in all, we taint 25,600 files, writing a total of 500MB of data. Table 6 shows the time taken for recovering from this attack by each of the versioning algorithms. Recovery is performed in two phases. In the first phase, once a malicious process has been identified, we traverse up that process's causal data graph to determine the root cause of the break-in. Backtracker [11] and Taser [7] perform a similar analysis to determine the cause. In the second phase, once we know the root cause of the attack, we propagate down the root process descendant tree to identify potential victims and recover them to a version just before the malicious process tampered with it. The recovery times that we report here are the times of the second phase. The results show that the recovery times are proportional to the amount of causal and versioning data stored. CA has the best recovery time and ALL has the worst recovery time. This is despite the fact ALL recovers the same amount of data as other algorithms and reads roughly the same amount of data as OC and GF to perform the recovery. ALL stores more versioning information than the other algorithms. Hence the required recovery data is spread over a much larger area on the disk. In turn, the recovery process has to perform more seeks to recover the same amount of data.

To answer the second question, we wrote a benchmark that simulates the Apache vulnerability scenario described in Section 1. The program first creates 50 files and then performs the following action in a loop 50 times. In each loop it writes 8KB to each file from start to end. Every n^{th} iteration (4^{th} in our implementation), the program forks a helper program that reads a byte from each of the 50 files and communicates the character to the main program via a pipe. This simulates the behaviour of a web server opening a new connection on a socket. In the causality based algorithms, once the main process reads from the pipe, it is a causally different version as it has read data from a new source, i.e., a new process. Hence any writes the main process performs after that creates a new version of the file. For this workload, OC does not copy any data in its version files; all the files were just created, so it considers all the writes to happen to version 1 of the files. CA and GF copy data to the version files on the iterations during which the the parent spawns a child and reads from the `pipe`. This stores 470.5MB of version data. The ALL algorithm copies data on every write and this adds up to around 2GB of data. The amount of version data is shown in Table 7.

We then recover versions at various intervals to get a sense for how expensive it is to go back further in time in each algorithm. There are 12 causality events in all, corresponding to the number of times a child is forked. We measure the time taken to recover data to a state before the 11^{th}, 7^{th}, and the 3^{rd} event. The 11^{th} corresponds to recovery close to the latest version, the 7^{th} corresponds to recovery two thirds of the way back, and the 3^{rd} corresponds to recovery close to one third of the way back. The results of this benchmark are shown in Table 7. With OC, there are no intermediate versions, so it cannot recover anything useful. CA and GF can both recover to a correct version and they both take the same amount of time to recover. They also read only the exact amount of data to be recovered. ALL, however, takes at least 9 times longer than CA and GF to perform recovery, because it has more data and has to search through a large amount of data to rebuild the correct version. Further, ALL has many more false positives that it has to filter before deciding on the version to recover. CA and GF have only one version to choose. Since ALL has been versioning continuously, one version of a process has multiple

children. For example, the 3^{rd} causal event has 41,000 children from which it has to narrow down to 50 versions. CA and GF have only 50 children for that causal event. Note that the numbers presented in the table do not include the time used to identify the version to recover.

6.3 Results Summary

In most cases, CA introduces little overhead relative to OC, yet it provides versions in cases where OC fails to do so. GF performs comparably in many cases, but sometimes imposes high run time overheads. Version-on-every-write practically always performs poorly both in terms of space and elapsed time. For recovery, however, CA and GF can indeed be a big win in terms of both the time to recover a particular version and the amount of data lost.

7 Related work

Several prior research projects have built versioning systems. We categorize these systems by the versioning algorithm that they use and discuss each class in turn. We begin with the version-on-every-write systems. CVFS [24] was designed with security in mind. Each individual write or small metadata change (e.g., atime updates) is versioned. The research focuses on methods to store and access old versions efficiently. We adopted the CVFS approach of using a journal to store old version data. Wayback [3] is a user-level versioning file system built on the FUSE framework. On a `write` call, Wayback logs the data being overwritten to an undo log before completing the write. Our version file format is similar to that of Wayback, but Wayback versions on every write while we version more selectively. The Repairable file system (RFS) [29] has functionality closest to ours. They record both causal data and save versions. They, however, collect causal data and data blocks separately, thus preventing them from taking advantage of causal information to version more selectively, leading to versioning on every write. They also have to reconcile the causal and versioning data using timestamps as they collect them separately.

Now we discuss systems that use open-close versioning. Elephant [18] is a versioning file system implemented in the FreeBSD 2.2.8 kernel. Their research focus is on providing users with a range of version *retention* policies. Versionfs [10] is a stackable versioning file system. Versionfs allows users to selectively version files and is focused on the ability to set space reclamation and version storage policies for files. Retention/reclamation policies are complementary to our work.

As we discussed in section 1, snapshots are another approach for versioning where an image of a file system is made periodically. Systems with snapshot functionality include AFS [13], Plan-9 [16], WAFL [9], [6], Venti [17], Ext3COW [15], Thresher [21], and Selective versioning secure disk system [27]. Skippy [20] proposes metadata indexing schemes that can be used to quickly lookup previous snapshots of a database.

Several systems have used causal data to provide various functions. The Taser intrusion recovery system [7] logs all system calls and their arguments. In the event of administrative errors or intrusions, they perform causal analysis on the logged data to determine the actions that need to be done to recover the system. They explore various algorithms and policies that can be used to determine the exact operations to be performed during recovery. We can leverage all of these algorithms and policies in our work, applying them in an online setting. As future work, they plan to integrate their work with versioning file systems to reduce the disk space requirements and to improve scalability. Our work has continued where Taser stopped and has taken a step further by integrating both causal and versioning systems. BackTracker [11] logs all system calls and in the event of an intrusion, performs causality analysis to determine the root cause of an intrusion. Autobash [26] is a configuration debugging tool that leverages causal information to limit the amount of testing required.

Chapman et.al. [5], explore techniques for causal data pruning. Their approach for pruning is to remove duplicates (which we already perform) and factor out common subtrees in causal graphs. Another approach for pruning causality could be to merge the causal information of deleted temporary files into their causal ancestors. Space can also be reclaimed by deleting the versioning data of temporary files, where temporary files are intermediate nodes in a causal graph.

Finally, a number of versioning algorithms have been explored by the object oriented database (OODB) community. These algorithms are focused on aspects that are particular to OODBs such as "how to propagate version changes of sub objects to composite objects?", "how to present a consistent view in the face of updates to different objects?" [4], "how to version classes as they change" [28], etc.

8 Conclusions

Combining versioning and causal relationship data offers powerful capabilities above and beyond what each kind of system can do in isolation. Causality-based versioning ensures that we create meaningful versions of objects, facilitating better recovery from data-corrupting activities under concurrent workloads. While versioning introduces overheads between 1% and 25%, adding causal collection on top of versioning adds only an additional

5–6% overhead. The Cycle-Avoidance algorithm, which restricts itself to considering only per-object, local information during online operation provides superior versioning and recovery, at cost comparable to open-close.

Providing versioning in the context of PASS opens up future research possibilities in the areas of reproducibility and archival. PASS did not previously provide the ability to reproduce objects on the system, because they do not preserve all the necessary data. However, with versioning, the necessary data do exist. Versioning also produces objects that can easily be archived, and PASS provides the provenance to accurately describe those objects.

9 Acknowledgments

We thank Ethan Miller, our shepherd, and Margo Seltzer for repeated careful and thoughtful reviews of our paper. We thank Erez Zadok, Shankar Pasupathy, Jonathan Ledlie, and Uri Braun for their feedback on early drafts of the paper. We also thank Uri for validating the CA algorithm in our user level simulator. We thank the FAST reviewers for the valuable feedback they provided. This work was partially made possible thanks to NSF grant CNS-0614784.

References

[1] ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, E. W., AND LIPMAN, D. J. Basic local alignment search tool. *Molecular Biology 215* (1990), 403–410.

[2] BRAUN, U., GARFINKEL, S., MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., AND SELTZER, M. Issues in automatic provenance collection. In *Proceedings of the 2006 International Provenance and Annotation Workshop* (May 2006).

[3] BRIAN CORNELL AND PETER DINDA AND FABIN BUSTAMANTE. Wayback: A User-level Versioning File System for Linux. In *Proceedings of the USENIX 2004 Annual Technical Conference, FREENIX Track* (2004).

[4] CELLARY, W., AND JOMIER, G. Consistency of versions in objects-oriented databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases* (1990).

[5] CHAPMAN, A. P., JAGADISH, H. V., AND RAMANAN, P. Efficient provenance storage. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 993–1006.

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

[7] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The Taser intrusion recovery system. In *SOSP* (2005).

[8] HALCROW, M. A. eCryptfs: An enterprise-class encrypted filesystem for linux. *Ottawa Linux Symposium* (2005).

[9] HITZ, D., LAU, J., AND MALCOLM, M. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference* (January 1994), pp. 235–245.

[10] K. MUNISWAMY-REDDY AND C. P. WRIGHT AND A. HIMMER AND E. ZADOK. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)* (March/April 2004).

[11] KING, S. T., AND CHEN, P. M. Backtracking Intrusions. In *SOSP* (Bolton Landing, NY, October 2003).

[12] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. In *the 12th Annual Network and Distributed System Security Symposium* (2005).

[13] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles* (1991).

[14] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*.

[15] PETERSON, Z., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage 1*, 2 (2005), 190–212.

[16] QUINLAN, S. A Cached WORM File System. *Software – Practice and Experience 21*, 12 (1991), 1289–1299.

[17] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies* (January 2002), pp. 89–101.

[18] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R., AND OFIR, J. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (December 1999).

[19] SHAH, S., SOULES, C. A. N., GANGER, G. R., AND NOBLE, B. D. Using provenance to aid in personal file search. In *Proceedings of the USENIX Annual Technical Conference* (2007).

[20] SHAULL, R., SHRIRA, L., AND XU, H. Skippy: a new snapshot indexing method for time travel in the storage manager. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA).

[21] SHRIRA, L., AND XU, H. Thresher: An efficient storage manager for copy-on-write snapshots. In *Proceedings of the Usenix Annual Technical Conference* (Boston, MA, May 2006).

[22] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A survey of data provenance in e-science. *SIGMOD Rec. 34*, 3 (2005), 31–36.

[23] SOMAYAJI, A., AND FORREST, S. Automated Response Using System-Call Delays. In *USENIX Security Symposium* (2000).

[24] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata Efficiency in Versioning File Systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (March 2003), pp. 43–58.

[25] Apache httpd 1.3 vulnerabilities. `http://httpd.apache.org/security/vulnerabilities_13.html`.

[26] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. Autobash: improving configuration management with operating system causality analysis. In *SOSP '07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), ACM, pp. 237–250.

[27] SUNDARARAMAN, S., SIVATHANU, G., AND ZADOK, E. Selective versioning in a secure disk system. In *Proceedings of the 17th USENIX Security Symposium* (July-August 2008).

[28] TALENS, G., OUSSALAH, C., AND COLINAS, M. F. Versions of simple and composite objects. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1993).

[29] ZHU, N., AND CHIUEH, T.-C. Design, implementation, and evaluation of repairable file service. In *The International Conference on Dependable Systems and Networks* (2003).