Versioning Extensions for Linux

J. Adam Butts and Paramjit Singh Oberoi
Computer Sciences Department, University of Wisconsin Madison
{butts,param}@cs.wisc.edu

Abstract

We describe the implementation of a dynamically loadable kernel module that adds versioning capabilities to the Linux operating system. The versioning system provides per-directory policies to control the creation of versions, the maximum number of versions per file, and the access permissions of version files. The module enables versioning over any file system that does not restrict the format of filenames and maintains forward and backward compatibility with the non-versioned file system. The module replaces system calls that can modify or delete files with ones that create new versions prior to execution of the original system call. The modified system calls present the same semantics to calling programs, making the presence of versioning undetectable. We show that the overhead of versioning is less than 20% for the open system call and less than 40% for the faster close system call. A cache of directory policies significantly reduces the overhead due to versioning during the open system call. Producing a new version results in the system call that causes the versioning to incur a large and file size dependent penalty due to the copy operation.

1. Introduction

Most computer users are familiar with the frustration of deleting or modifying a file and later realizing that important information was lost. This can be particularly serious if those files happen to contain source code that is a part of a big software project. Various tools (e.g. CVS [1]) exist that enable users to keep track of—and rollback if necessary—the changes made to a group of files. Most such tools available today require a user to do something special whenever the file is modified. Since this is usually inconvenient, these tools are only used in cases where it is absolutely necessary to be able to access old versions of files.

A versioning file system provides the same functionality in a completely transparent way. Users or application programs read and write to files in the same way as on non-versioning file systems.

The file system itself tracks changes to files and provides ways to access old versions of files, rollback changes, and control policies that decide various parameters related to versioning.

Versioning file systems have been around for some time. The TENEX operating system (a commercialized version of TOPS-10) first included versioning in its file system [2]. File names could specify a version number explicitly or the system would assume an appropriate default (e.g. the latest version for a file open and the oldest version for a delete). A method of explicit specification of the oldest or most recent versions without the number was also provided. VMS, very widespread during the reign of the VAX, implemented a similar versioning system [3].

The Cedar operating system from Xerox PARC also implemented versioning [4]. Unlike TENEX and VMS, files in Cedar are immutable. Furthermore, filenames are unique across the file system. These properties reduce the problem of maintaining consistent copies of files across a distributed file system to a trivial task. The implementation of such a scheme uses file version numbers to provide unique names to different instances of a file. Because files may not be modified, new files, identical in name to their source files except for the version number, are created when files are written. Cedar also provides for an automatic per file limit on the number of versions to reduce the pressure on storage space.

The behavior of the versioning extensions we will describe is externally similar to those in prior systems. The actual implementation choices were driven by a number of design goals, the most important of which was avoiding kernel modifications. Users can install and use the versioning features without the need to change their operating system kernel. We implemented a dynamically loadable kernel module that adds the versioning features without requiring kernel recompilation or even remounting of file systems. A related goal was ease of use of the versioning features. Installation or removal of the module and the associated versioning capabilities is effected by a single command. Several utilities were also written to aid in version and policy file management. Flexibility is important if the versioning features are to achieve widespread use. Users have precise control over whether or not versioning is performed, what circumstances cause a file to be versioned, the protection of version files, and the number of versions to be maintained. The infrastructure also allows for straightforward addition of new policies. Another important goal was complete forward and backward compatibility between existing file systems and those with versioning extensions. Because versioning may be dynamically added and removed from the operating system, it is critical that files and directories be accessible with or without the module loaded regardless of the module's presence or absence during their creation. All data recovery and diagnostic utilities (like `fsck`) will continue functioning normally and will be able to recover all the data (if not more) after a crash that they could have recovered without versioning.

The versioning extensions were developed on an Intel Pentium Pro based computer running GNU/Linux 2.2.2 (hereafter referred to as Linux). Linux was selected for three main reasons. First, the entire source code of the kernel as well as the device drivers, file systems, and other associated components is public [5]. Second, Linux supports kernel modules that can dynamically extend the functionality of the operating system. Finally, extensive documentation on the operation of the kernel [6] and kernel modules is readily available [7]. While changes to the system call implementation within the kernel could require changes to the module, this code is fairly mature and unlikely to change. Therefore, the module will not require extensive changes to support more recent kernels (no changes were required to support the latest version available at the time of publication, 2.2.7). Also, there is no architecture dependent code within the module, allowing it to be easily compiled for alternate platforms.

The remainder of the paper describes the versioning extensions in detail. The next section describes the interaction of the versioning module with the Linux kernel and the user. Section 3 describes the implementation and limitations of the versioning features. Performance data are presented in section 4. Conclusions and opportunities for further work are presented in the final section.

2. Design Definition

This section describes the external interfaces of the versioning module, both with the user and with the Linux kernel. First, the interface between the kernel and the module is addressed. The effects of versioning from the user's viewpoint are considered next. The exact behavior of versioning is determined by user-defined policies, which are discussed in section 2.3.

2.1. Kernel Interface

Operating systems that support versioning have this functionality built into the file system and kernel from the beginning. When adding versioning to an existing operating system, in this case Linux, there are two logical places that the functionality can be added without requiring modification of the kernel. The versioning extensions may be inserted between the kernel and file system or between user programs and the kernel.

The first approach involves writing a virtual file system that logically presents the view of a versioned filesystem to the higher layers of the kernel while using a conventional non-versioned filesystem to do the actual disk I/O. This layered approach is used in the UMSDOS filesystem, which implements a UNIX-like file system over an MS-DOS FAT file system. By mounting an existing file system through the virtual versioning layer, versioning may be transparently added to any file system. This type of design also ensures complete backward compatibility because all disk operations are performed using the underlying file system.

The other possible approach is to modify the interface between application programs and the operating system. Applications access the file system through a limited set of well-defined system calls. All file system related system calls can be intercepted and translated to an appropriate sequence of calls that can maintain older versions of modified files. Most operating systems (including Linux) provide a way to replace the system calls dynamically, obviating the need for kernel modifications.

The first approach makes versioning a property of a specific file system: versioning will only occur if the file system is mounted through the virtual versioning layer. This can be confusing to users who see a single namespace. It is unreasonable to expect users to adjust to the fact that files in some areas of the file namespace may be versioned while those in other areas cannot. Also, versioning is a high level activity: it is defined in terms of processes and files. The operations that must be supported by a file system, however, are of a much lower level. This semantic mismatch complicates the implementation of versioning at the file system level since the higher level information is unavailable. Most importantly, file systems in Linux share data structures with the kernel. Unfortunately, this makes it extremely difficult to implement versioning in a completely safe manner.

We have chosen the second approach for adding versioning functionality. The versioning extensions are implemented as a single, dynamically loadable module. The module replaces all system calls that can modify a file by functions that behave exactly the same way as the original except that they make a copy of the file before modifying it. Because all file operations require a system call, the versioning layer is guaranteed notification prior to any update of a file. Also, the file manipulations required to create new versions and maintain per directory policies are straightforward at the system call level. Forward and backward compatibility are achieved since policy and version files are regular files on the file system. File systems that have these special files (versioned file systems) are still completely usable without the module, and non-versioned file systems behave normally when the module is installed.

2.2. User view

Version files are exact copies of their source files created automatically by the module prior to the modification of those source files. They are stored as regular files in the same directory as the original file. The module uses the name `.filename,n` for the n^{th} version of `filename`. `filename` may optionally contain additional dots, commas, or numbers. The prepended dot is a UNIX convention to denote a hidden file. This prevents the version files from cluttering directory listings generated by `ls`, for example. A new version file receives a version number one more than the highest existing version number for the same file, regardless of numbering gaps that may exist. If no prior versions exist, the numbering starts at zero.

Other than the name change, the version files behave exactly like regular files; they may be read, modified, and deleted. Modification of a version file results in hierarchical numbering. For example if the file `.report,4` is modified, version files named `..report,4,0`, `..report,4,1`, etc. will be created. The only difference in the behavior of version files and regular files occurs if the user specifies that new versions are to be created on file deletion: deleting a version file will not cause a new version of it to be created.

To aid the user in managing the version files and the version policies themselves, four small utilities were also developed with the versioning module. Three of these are used for version file management, while the fourth is the utility used to define versioning policies. The version management tools are `undelete`, `revert`, and `purge`. `undelete` is the simplest of these; it causes the latest version of a non-existent file to be renamed to the original filename. `revert` allows the user to cause an existing file to revert to a previous version selectable by absolute or relative version number. `purge` is provided to monitor the space taken by version files and remove or renumber them. The policy definition tool `setpol` is used to display or modify the policies in a given directory.

2.3. Policies

Each directory has a special file named "`.version`" that contains the user-specified versioning policies for that directory. Files in a directory that does not have a `.version` file are not versioned. A full list of the policies that may be specified is presented in Table 1. Each of these policies is detailed below.

Policy	**Explanation**	**Example**
VersionOnWrite	Create new versions on modification	Yes/No
VersionOnUnlink	Create new versions on deletion	Yes/No
MaxVersions	The maximum number of old versions	1, 2, 10, ...
RdOnly	Old versions are made read-only	Yes/No
ExcludeRegEx	Files to be excluded from versioning	`*swp | *.o | .*`
IncludeRegEx	Files to be versioned even if they match *ExcludeRegEx*	`*thesis*`

Table 1: Versioning Policies

The *VersionOnWrite* and *VersionOnUnlink* policies specify which file operations will result in the creation of a new version. The *VersionOnWrite* flag enables version creation whenever a file is modified. Modification includes writing or appending data or truncation of the file. Setting *VersionOnUnlink* will cause a new version to be created when a file is deleted. The separation of these enabling flags allows for simple undelete functionality without versioning files which are changed.

Automatic creation of new files from versioning could rapidly take up all available free space in a file system. Requiring the user to monitor and purge old versions is unreasonable. The policy *MaxVersions* limits the number of old versions of each file. If creation of a new version would cause the number of versions to exceed *MaxVersions*, the oldest version is deleted. User management of version files is also permitted.

The *RdOnly* policy is a safety feature to prevent accidental deletion or execution of old versions. Version files created with this policy enabled have all write and execute bits in their access permissions cleared. While the files will still be subject to automatic purging under the *MaxVersions* policy, manual deletion, modification, or execution will require the user to change the permissions explicitly. When *RdOnly* is not enabled, version files inherit the permissions of their source files.

A feature that we found absolutely necessary to make use of versioning is the ability to exclude some files from being versioned. Most editors and compilers create many temporary files that the user does not care about. Versioning these files would waste space on the disk and unnecessarily clutter the file system's namespace. Object files are another class of files for which most users would like to disable versioning. The user can specify the excluded files using a restricted set of regular expressions using the *ExcludeRegEx* policy. The regular expression syntax allows for multiple patterns that may each optionally utilize greedy wildcards, single character wildcards, and single or multiple digit matches.

Finally, to allow for the versioning of specific files, an *IncludeRegEx* is also provided. A file will be versioned if its filename matches the *IncludeRegEx* even if it would match the *ExcludeRegEx* (provided the other requirements are met; see Figure 1). One practical use of such a policy is realized by setting the *ExcludeRegEx* policy to match all files: only those files that match the *IncludeRegEx* will be eligible for versioning.

3. Implementation

The internal structure of the versioning module is described in this section. The module insertion and removal process is addressed first. Detailed descriptions of modifications to the system call are presented next. Section 3.3 presents the module's internal data structures. Version file creation and deletion are discussed in section 3.4. Finally, the limitations of the current implementation are addressed in section 3.5.

3.1. Module Insertion and Removal

Module insertion is a straightforward process. Linux maintains a table of pointers to each system call. The pointers corresponding to functions that will be replaced are copied and pointers to the modified system calls are inserted. The Linux kernel is locked to ensure that processes will see the new system calls appear together.

While it may seem that module removal is simply the restoration of the original function pointers, removal is complicated if there are multiple modules in place that have installed new system calls. If the module removal order is not the reverse of the insertion order, invalid function pointers can be placed in the system call table. Assume, for example, that the original system call `open()` was replaced first with `openA()` by module A and subsequently with `openB()` by module B. If module A

System Call	Reason for Replacement
`open(), creat()`	Versioning policy management, files may be truncated
`close()`	Versioning policy management
`write(), writev(), mmap()`	Files may be modified
`unlink(), rename()`	Files may be deleted
`truncate(), ftruncate()`	Portions of the files may be deleted

Table 2: System Calls Replaced

is removed, it will replace `openB()` with `open()`, canceling the effect of module B without removing it. When module B is removed, `open()` is replaced with `openA()`, a now invalid function.

One solution to this problem is to require a system reboot to remove the module. Each module has a use count to prevent a module in use from being removed. By incrementing the use count on insertion, removal of the module is prevented. While this is a simple solution, it is overly conservative. Provided that the functions installed by the module are still in the system call table, the removal of the module is guaranteed to be safe. The versioning module may be removed if this condition is satisfied.

3.2. System Calls

The versioning module replaces all system calls capable of deleting or modifying a file. Enforcement of the versioning policies requires modifications to the `open()` and `close()` system calls. Table 2 lists all of the system calls replaced by the module. The specific modifications are detailed in this section. Stevens' book on UNIX programming [8] was invaluable in determining the nuances of the various system calls.

3.2.1. `open(), creat()`

The `open()` system call determines if the file being opened will be versioned when modified. If the file does not have to be versioned, control is transferred to the original system call. Otherwise, an entry is added to a global list of open files eligible for versioning. This list is called the open file table and is described in detail in section 3.3.1.

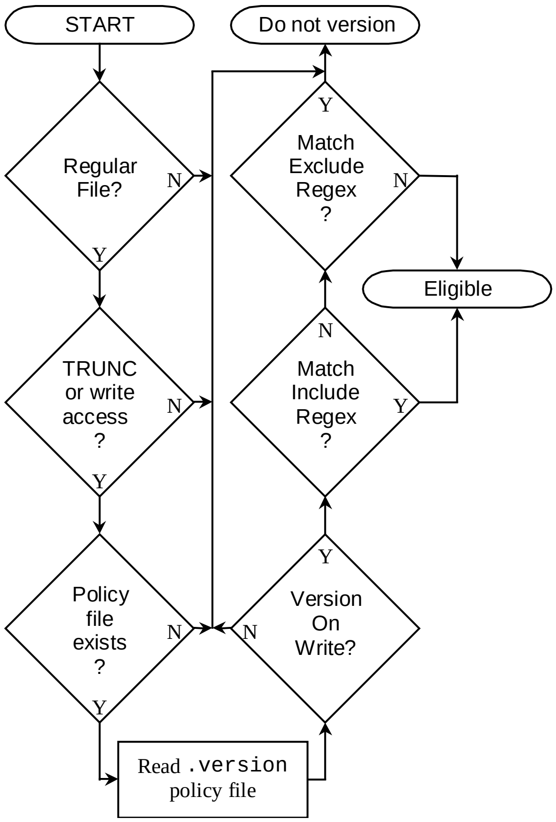

Figure 1: Determining Versioning Eligibility

The decision algorithm used by `open()` is shown in Figure 1. Provided the file is a regular file (i.e. not a pipe or device, for example) opened for writing, the policy file in the appropriate directory is read. If the policy file does not exist, the file is not versioned. If the directory policies specify *VersionOnWrite*, the filename is checked against the *IncludeRegEx* and *ExcludeRegEx* regular expressions. The file is eligible for versioning if either the filename matches the *IncludeRegEx* or it does not match either expression.

An additional subtlety of the `open()` system call is that a flag may be set to truncate the file. If this flag is set and the file is eligible for versioning by the above algorithm, a new version is created prior to the truncation. A single `open()` can result in at most one version to be created prior to the matching `close()`. Therefore, if the version is created due to file truncation on open, subsequent writes to the open file will not result in additional versions.

`creat()` is equivalent to `open()` except the file is truncated if it exists or is created otherwise. For versioning purposes, `creat()` behaves as `open()` with the file truncation flag set.

3.2.2. `close()`

During `close()`, the open file table is checked for an entry corresponding to the file being closed. The entry will be removed if the usage count of the file will be zero after the close is performed. This ensures that an entry will remain in the open file table as long as the file may be modified, even if the original file descriptor is closed. Such a situation can occur if the file descriptor is duplicated, for example. Control is then transferred to the original `close()` function.

3.2.3. `write(), writev(), ftruncate()`

These functions operate on open files and either cause modification or truncation of the file. On each of these calls, the open file table is searched for the file being modified. If a reference to the file is found and *VersionOnWrite* applies to the file, a copy is made prior to allowing the write or truncation to proceed. The copy is performed only the first time that a file is modified after being opened. Subsequent writes to the file do result in additional copies until it is closed and opened again.

3.2.4. `mmap()`

The `mmap()` system call allows a program to map a file into its virtual memory and then access it using normal memory references. It is difficult to detect when a file mapped into memory is written. The only way of detecting these writes is to mark the pages associated with the file to read-only so that any write causes a protection fault. The fault handler would then be required to recognize that the write might proceed after the unmodified file is copied. Modification of the fault handler is difficult from a kernel module. Therefore, the current implementation pessimistically copies the file if it is mapped into memory in any mode that allows writing. Thus, `mmap()` is handled exactly like `write()` and `writev()`.

3.2.5. `truncate()`, `unlink()`, `rename()`

The `truncate()` system call changes the length of a file, potentially causing loss of data. `unlink()` is used to delete a file while `rename()` can also cause deletion if the new name corresponds to an existing file. All of these functions operate on unopened files and therefore the open file table can not be used to determine whether versioning should occur. Instead, almost the same algorithm used by the `open()` system call is applied except that the *VersionOnUnlink* flag is checked for `unlink()` and `rename()`. Version file creation for `unlink()` and `rename()` do not require the file to be copied; the file is simply renamed to the appropriate version file.

3.3. Internal Data Structures

The versioning module maintains two data structures to reduce the overhead introduced by versioning. The directory policy cache maintains the contents of recently read policy files. The open file table contains information about the applicable policies and state of open files eligible for versioning.

3.3.1. Open File Table

The open file table contains one entry for each open file eligible for versioning. Since the number of open, eligible files may be large, the table is hashed to provide for quick access. The index is a function of the kernel file structure pointer. This makes an ideal index because a new distinct file pointer is assigned each time a file is opened. Therefore, the same file opened from different directories (because of a link) that have different policies will be handled in each directory correctly since it can have two different entries in the open file table. Entries are inserted into the table on an `open()` or `creat()` and removed on a `close()` provided the usage count on the file will become zero (see section 3.2.2).

Entries in the open file table contain the versioning policies that apply to that file and some state information. The policy information comes directly from the appropriate policy file or from the policy cache (described below). A dirty flag is set when a file is first modified and versioned. Subsequent modifications use the dirty flag to determine that a new version does not need to be created. Another flag is set if the open file corresponds to a policy file. The use of this flag is discussed in the next section.

3.3.2. Directory Policy Cache

To avoid having to read directory policies each time a file is opened, a cache of policies is maintained by the versioning module. The policy cache is a 64-entry direct-mapped cache indexed with a portion of the inode number of the directory. The complete inode and device number are used as a tag since this combination is guaranteed to be unique. System calls which must read directory policies (e.g. `open()` and `unlink()`) only read the `.version` file if the policies are not present in the cache. If a read is required, the cache is updated with the recently read policies. For directories that do not contain a `.version` file, a valid entry is cached that disables versioning within that directory.

System calls which can modify the `.version` file must be handled specially to avoid inconsistency between the cache and the policy file. Upon opening a policy file for writing, the corresponding cached policies are read (if not already in the cache) and pinned to maintain consistent behavior during the update. No other entry may evict the pinned entry until the policy file is closed. A special entry is inserted into the open file table denoting an open policy file. When the policy file is closed, `close()` invalidates the pinned cache entry if the file was modified. Thus, the new policies will be read into the cache the next time that they are required.

The effect of cache size on the miss rate was not examined. The cache need only hold the working set of directories in use on the system. For a single user system, we expect this number to be small since a user typically operates on files in a sub-tree of the user's home directory. For multi-user systems, we expect the cache size requirements to scale approximately linearly with the number of users. A policy cache entry occupies only sixteen bytes in the current implementation, making a large cache fairly cheap.

3.4. Version File Management

Version creation involves copying an eligible, unmodified file to a version file prior to the first modification. The version filename is derived from the original name as described in section 2.2. Because the entire file is copied, the cost of the first modification is file size dependent and can be large. A copy-on-write scheme to share common portions of the file would greatly reduce this first write penalty, but would require modifications at the file system level. Modifications at such a level are not possible from a module and would introduce a dependency on the type of the underlying file system.

Each time a new version of a file is created, the number of old versions in the directory is compared against the policy limit *MaxVersions*. Old versions are deleted from the lowest version number up until the total number is *MaxVersions*. The versioning module assumes that any file named `.filename,n` is the n^{th} version of the file `filename`. The actual version numbers are not involved in enforcing the version limit—only the number of old versions is important. Therefore, policy enforcement is handled correctly even if a user decides to delete selected old version files manually (e.g using `purge`).

3.5. Limitations

The current implementation of the versioning module suffers from several limitations. Most notable are those imposed by the attributes of Linux kernel modules. Both of the major limitations in this class have already been addressed: the pessimistic handling of `mmap()` in section 3.2.4 and the inability to implement a copy-on-write scheme discussed in the previous section.

Another limitation is that the versioning module will not currently run on a multiprocessor. No specific synchronization was implemented to protect the data structures of the versioning layer from multiple simultaneous accesses. Therefore, it is possible for race conditions to arise if it is run on a multiprocessor. Support for multiprocessor machines is still in its infancy in the Linux kernel and we expect the mechanisms related to locking to change in the next few releases. The granularity of locks in the kernel also needs to be improved: at present there is just one global lock on the entire kernel.

Finally, although designed to be independent of the underlying file system, the module assumes that the underlying filesystem has no restrictions on filenames. Therefore, it will not be able to function over file systems that restrict the size and format of filenames (e.g. FAT).

4. Performance Evaluation

4.1 Methodology

We measured the time taken by `open()`, `close()`, and `write()` system calls with and without versioning for varying file sizes. The measurements were made using the clock cycle counter of the Intel Pentium Pro microprocessor. Making sure that the required data was cached in the operating system's disk buffers minimized the effect of the disk performance on these results. The performance measurement programs were run as real-time processes with the highest possible priority to prevent the benchmark from being affected by context switches.

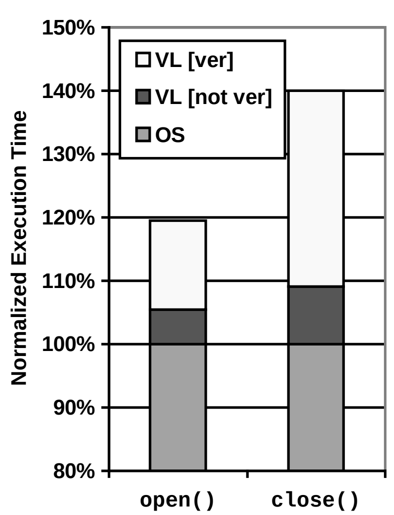

Figure 2: Opening and Closing Files

Each measurement was repeated a large number of times (10,000 for `open()` & `close()`; 100 for `write()`), and the arithmetic mean of the individual measurements was reported as the final result. To ensure maximal use of the cache, the first few measurements for each file were ignored while calculating the mean.

4.2 Results

Our primary performance goal was that file access time in directories that have versioning turned off should not change significantly. Figure 2 shows the time taken

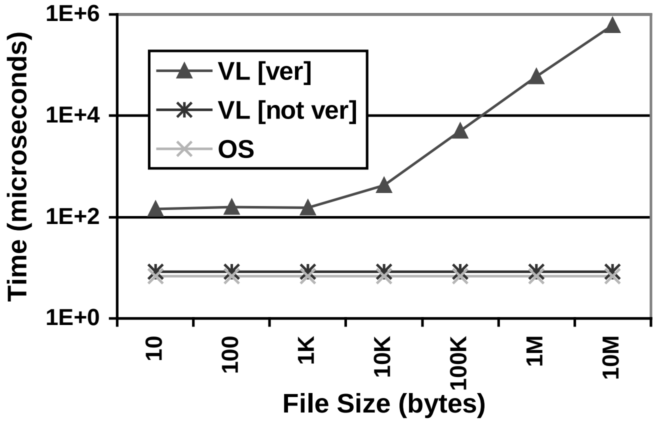

Figure 3: Versioning Overhead on `write()`

by the `open()` and `close()` system calls without the module (labeled OS), with the module loaded but versioning turned off (labeled VL[not ver]), and with versioning turned on (labeled VL[ver]). The overhead when versioning is disabled is minimal. When versioning is enabled, the time taken by `open()` increases by about 20%. The `close()` system call suffers a relatively higher increase of 40%. However, that is because the close system call takes significantly less time than `open()`. The time taken to open and close a file was independent of the file size.

The major performance impact of versioning occurs on the first write to a file. If that file is being versioned, a copy of the file has to be made before the actual write is allowed to happen. Figure 3 shows the time taken to copy the file as a function of the file size. The graph shows that copying a file is a very expensive operation—the time taken for the `write()` system call increases by a factor of ten for very small files. The time required increases linearly with the number of disk blocks in a file for large files. However, for file sizes up to 100Kb, the copying time is under one second. Since most files that users usually wish to version (e.g. source code or configuration files) are small, this overhead is acceptable.

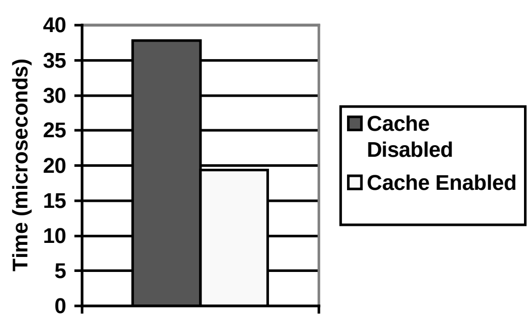

Figure 4: Time taken by open()

We also measured the overhead of versioning with the version policy cache disabled to evaluate the effectiveness of the cache. While the operating system will probably cache the version policy files that are frequently read in its own buffers, we found that caching the policies in the module itself significantly reduced the overhead in the `open()` system call. Without a policy cache, the open system call has to open two files: the actual file being opened and the corresponding `.version` file. This increases the time required for opening a file by approximately 100%.

5. Conclusions and Future Work

We have shown that versioning capabilities can be added in a file system-independent way to the Linux kernel. No low-level changes to the file system are required and complete forward and backward compatibility can be achieved. We have implemented this approach and shown that the performance degradation because of this added functionality is within acceptable limits. The addition of a version policy cache was found to significantly reduce the overhead of the open system call. We also designed and implemented several tools to manage old versions and control versioning policies.

One interesting direction for future work is storing older versions as differences or as compressed files. This would significantly reduce the space overhead of versioning and so allow versioning at a finer granularity than open-close pairs. New versioning policies can be added that will allow the user to control the storage format of old versions.

A major problem in our current implementation of versioning is the latency of the first write to a file that is being versioned. However, the write does not have to occur immediately if the operating system is able logically present the view of the changed file to the application. The actual operations relating to versioning can happen in the background and the application need not see long latencies.

Finally, multiprocessors present a challenge. There are many race conditions that can occur when multiple processors access the same file simultaneously. While it is trivial to solve this problem by restricting the number of processors that may execute the code related to versioning, a more efficient solution is desirable.

6. References

[1] Dick Grune, Brian Berliner. Concurrent Versioning System (CVS). April 1989.

[2] Daniel G. Bobrow, et.al. TENEX, a Paged Time Sharing system for the PDP-10. *Communications of the ACM,* March 1972.

[3] Kirby McCoy. VMS File Systems Internals. *Digital Press*.

[4] Michael Schroeder, David Gifford, and Roger Needham. The Cedar File System. *Communications of the ACM,* March 1988.

[5] Linus Torvalds, et al. The Linux 2.2.2 Kernel source code, 1999.

[6] Michael K. Johnson. Linux Kernel Hackers' Guide, 1996.

[7] Nikos Drakos. Linux Module Programming Guide, 1997.

[8] W. Richard Stevens, et.al. Advanced Programing in the UNIX Environment. *Addison Wesley Longman, Inc*. Feb 1992.