# Prism: Providing Flexible and Fast Filesystem Cloning Service for Virtual Servers

Xin Zhao[1], Kevin Borders[2], and Atul Prakash[2]

[1] Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
xinzhao@google.com
[2] University of Michigan
2260 Hayward Street
Ann Arbor, MI 48109-2121, USA
{kborders,aprakash}@eecs.umich.edu

**Abstract.** This paper describes a prototype virtualized file system, Prism, for supporting hosted servers and utility computing. Prism provides a filesystem service that allows lightweight creation of filesystems for new users from existing filesystems. All users' filesystems are mutable and yet isolated from each other. In our experiments, new filesystems can be created from existing ones in under one-fifth of a second. Prism is also designed to make centralized security-related services across multiple, similar filesystems more efficient. In particular, with Prism, tasks such as virus checking over multiple filesystem clones are much more efficient than scanning each user's filesystem independently. We describe the design of Prism and present performance results.

## 1 Introduction

One application scenario of hosted services and utility computing is to be able to provide remote users with dedicated data and computing facilities using centralized computing resources. This paper focuses on one aspect of the problem: providing dedicated filesystems to users on demand, as well as common filesystem-related services, such as on centralized virus scanning on users's filesystems in a lightweight way.

This paper describes a prototype virtualized file system, Prism, which supports multiple filesystems, where each filesystem can be assigned to a different user. Each user gets the illusion of having a full-fledged filesystem, which in principle, can include system files, applications, and user files, all under the control of the user. Prism provides an efficient filesystem cloning mechanism to create new filesystems from existing ones. A filesystem clone is semantically similar to a copy of the parent filesystem. Once created, it is independent of the parent filesystem. Subsequent changes to either one are not reflected in the other. Prism's mechanism guarantees isolation of users' filesystems, while providing very fast creation of new filesystems from existing ones. In our tests, new filesystems that are created from existing ones are usable within one-fifth of a second

V. Issarny and R. Schantz (Eds.): Middleware 2008, LNCS 5346, pp. 388–407, 2008.

and provide comparable performance to native `ext3` filesystem when the cloning is complete.

Prism is also designed to make centralized security-related services across multiple, similar filesystems more efficient. In particular, multiple filesystems can often be scanned collectively for tasks such as virus checking much more efficiently than scanning each user's filesystem independently. In our prototype setup, simulating a virus scanning task on all the files for eight cloned filesystems was approximately three times faster than doing eight individual scans.

Prism's mechanism for instantiating a new filesystem from an existing one provides a feature called *selective cloning*. In selective cloning, a user can request a clone of an existing filesystem, while excluding specified directories or files from being cloned (optionally replacing them with default substitutes.) We anticipate that this capability can be a useful feature in specialized scenarios. Consider a user Alice who is given a virtual machine running Linux on a hosted service provider, along with a dedicated filesystem that is provided by Prism. She wants to install a new software application that appears useful, but she is not sure if she should trust it and is not sure if it will be compatible with existing software. She decides to request the Prism's cloning service to provide her a clone of her filesystem, but excluding sensitive files such as her home directory, contents of /tmp and /var/log. This new filesystem can be used to provide her a testing environment that is very close to her current environment, but less susceptible to data theft, all within a few seconds.

Prism's cloning abstraction is semantically similar to making a copy of the entire filesystem, except it appears to be much faster to users. An end-user can get a usable cloned filesystem almost instantaneously, irrespective of the size of the cloned filesystem (either in number of files, depth, or total number of bytes). In addition, the filesystem cloning operation will not interrupt access to the parent filesystem.

Prism makes extensive use of copy-on-write at both file level and for blocks within files so as to use disk space efficiently when providing filesystem services for multiple users. The parent and cloned filesystems share data of unchanged files, which usually occupy a large portion of files. Furthermore, when a shared file is modified, an unchanged blocks continue to be shared.

Prism is currently in prototype stage. It has around 5000 lines of code. We have used Prism to host filesystems for multiple virtual machines. To evaluate Prism's cloning performance, we cloned a standard Fedora Core 4 distribution that consists of over 170K files and over 17K directories. The cloning operation itself was essentially an immediate operation from the perspective of the end-user, taking only 0.18 seconds to complete. After 0.18 seconds, both the parent and the cloned filesystem were completely accessible to end users. In terms of disk space, a clone took up about 1.3% of the space (77MB for the clone versus 6GB for the parent filesystem).

We also measured performance of a Prism-cloned filesystem on several workloads and compared it with solutions based on the `ext3` filesystem. On the Connectathon [1] benchmark and an Apache-build workload, the cloned filesystem's

performance was comparable with that of an ext3-based filesystem, with only a minor performance penalty. For scanning multiple cloned filesystems, Prism outperformed an ext3-based solution significantly because it was able to skip over the files that had not been modified since cloning. The performance advantage of Prism over ext3 went up as the number of clones was increased.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 illustrates the design of the Prism cloning. Section 4 presents evaluation results. Section 5 concludes this paper.

## 2 Related Work

Prism borrows ideas from existing filesystems with snapshot and versioning capability, such as WAFL [2], CVFS [13], VersionFS [9] and Ext3cow [10], but also introduces some differences. Like them, it makes extensive use of copy-on-write to help reduce overheads. In versioning filesystems, the notion of providing complete, dedicated filesystems for different users is usually missing. Instead, the assumption is that all versions are under one administrative control. In contrast, filesystem clones in Prism are all writable and isolated from each other; they are designed be exported to different users.

Several recent filesystems, such as Flexclone [4], VMFS [16], Parallax's filesystem [17], and ZFS [7] provide writable snapshots. Prism differs in a few ways in its design. These systems generally use block-level copy-on-write, where the file-level semantics are not available. This would make it difficult to exclude specified files or directories during a snapshot. In contrast, Prism is aware of filesystem structure and uses file-level copy-on-write. It is therefore trivial to selectively exclude directories from a snapshot or even graft part of one filesystem into the clone of another filesystem to compose a new filesystem. As we show later, file-level copy-on-write also permits more efficient central scanning across multiple filesystems.

Some systems, such as UnionFS [19], Ventana [11], Alcatraz [5], IFS [14], and Feather-weight Virtual Machine (FVM) [18], provide efficient cloning-like capability using a metadata manipulation technique. The key idea is to deploy a filesystem virtualization layer to manipulate the pathnames of files when a client or VM requests to access them. As shown in Figure 1, all other VMs are assumed to be created by cloning and sharing the base filesystem. If a user needs to change a shared file, these systems create a new copy in the writing user's

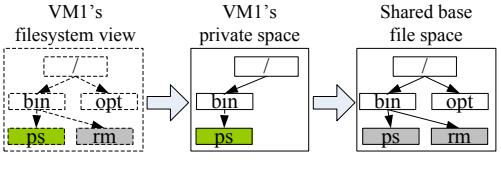


**Fig. 1.** Cloning via namespace manipulation

private space. Upon receiving a file request, the filesystem virtualization layer first checks the user's private space so that the private copy can override the shared copy.

However, cloning filesystem by manipulating pathnames is not as flexible as the mechanism used by Prism. Prism's design makes it easy to create clones of any user's filesystem, even the parent filesystem is a clone itself. In contrast, systems like FVM assume that only a base filesystem will be cloned. Furthermore, compared with normal filesystems, manipulating pathnames in private and shared spaces incurs higher lookup overhead to locate the right file corresponding to a given pathname. Prism does not introduce another pathname translating layer and thus achieves performance close to a filesystem without cloning support.

## 3 Prism Design

### 3.1 Background

Prism was designed by modifying the *ext3* filesystem and adding support for cloning and exporting any part of the filesystem to a user. Currently, Prism's filesystems are simply exported using NFS. In principle, Prism's filesystems could also be made available for access using other protocols such as Samba.

In the most general usage scenario, Prism exports a user's filesystem to the user with full read-write privileges. It is trivial to limit the user to read-only access to selected files that should not be updated by the user, if desired. We have previously proposed a server-side policy engine to do that in [21].

In the rest of the paper, we assume that each user is accessing the filesystem from a standard operating system. In our experiments, we emulated these users' operating systems using guest virtual machines that were hosted by a centralized server. To simplify terminology (since "user" can be an overloaded term), from now on, we will refer to user's operating systems as client virtual machines (VMs), even though users can access Prism filesystems over a network from standard operating systems as well.

As shown in Figure 2, Prism runs a modified ext3 filesystem called *pext3* to manage files for all the users. On the pext3 filesystem, Prism stores each client VM's files in a *fileset*. A Prism fileset is similar to a *volume* in AFS [3,8,12]. It

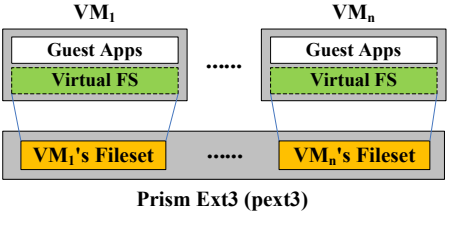


**Fig. 2.** Cloning via file sharing

is a tree of files and sub-directories on the physical repository managed by the pext3 filesystem. Prism exports a client VM's fileset as a virtual filesystem over NFS. In Prism, cloning a client VM's filesystem is accomplished by cloning the VM's fileset on the pext3 filesystem.

Prism provides three forms of cloning: basic cloning via file sharing, asynchronous cloning, and lazy cloning. We first describe the basic cloning mechanism in which the entire directory hierarchy is cloned. Then, we describe the asynchronous cloning mechanism that allows new filesystems to be usable almost immediately without cloning the directory hierarchy in entirety. After that, we describe lazy cloning, in which directories and files are cloned only as needed.

### 3.2 Synchronous Cloning Via File Sharing

As Figure 3 shows, Prism avoids copying files that are the same in a clone as in the parent filesystem. To clone a filesystem, Prism always starts from the filesystem's root directory to traverse the entire directory structure and clone the encountered filesystem objects that are not flagged as "nonclonable". For each clonable directory, Prism creates a new directory at the corresponding place in the clone. For a regular file, Prism clones it by creating a hard link to that file in the clone. In fact, all named files can be regarded as hard links in Prism. The name associated with a file is simply a label that refers the operating system to the actual data. More than one name can be associated with the same data. A hard link is essentially a directory entry that associates a file name with the actual data. By creating a hard link to the original file's inode, the cloned file shares the same content with the original copy without physically duplicating the data blocks. Copy-on-write is performed to create a new private copy if either the clone or its parent attempts to change a shared file. All subsequent modifications are applied to the new copy. As such, the isolation between the parent filesystem and its clone is still preserved. This cloning procedure is similar to the copying operation in conventional filesystems and thus very flexible. One can easily clone any selected part of a filesystem to a specified location.

The above solution is inadequate if support for hard links is required in users' filesystems. Figure 4 shows an example that helps illustrate this issue. Suppose a user clones a filesystem `FS1` to a new filesystem `FS2`. We refer to `FS1` as the "parent" filesystem and `FS2` as the "child" filesystem. In `FS1`, the file `/a/b` and `/x/y` are two hard links pointing to the same file on disk. When cloning `FS1` to

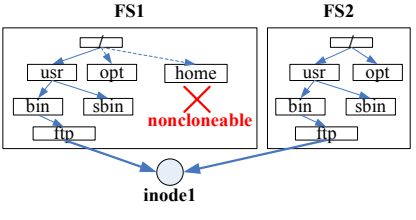


**Fig. 3.** Cloning via file sharing

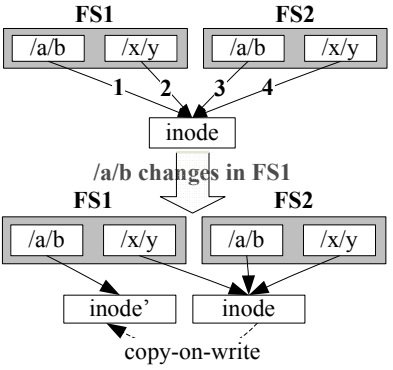


**Fig. 4.** Copy-on-write on an inode breaks the hard link semantics of a standard filesystem. If both FS1 and FS2 are fully copied filesystems, both /a/b and /x/y should point to the same inode even if the file content is modified.

`FS2`, Prism creates two hard links in `FS2` for `/a/b` and `/x/y`, respectively. Now there are four hard links point to a same file. We cannot preserve the Prism cloning semantics and the standard hard link semantics under such a filesystem structure.

Suppose a user writes the file `/a/b` in `FS1`. At this time, the file associated with `/a/b` is being shared by `FS2`. In order to preserve isolation between `FS1` and `FS2`, Prism duplicates the shared file to a new copy (represented by `inode'`), and adjusts the `/a/b` entry in `FS1` to point to the new copy. However, `/a/b` and `/x/y` in `FS1` now point to different files. This breaks the hard link semantics that can be preserved in a fully copied filesystem. According to the standard UNIX hard link semantics, though hard links have different names, data changes made through any hard link will affect the actual data and are immediately visible to other hard links pointing to the same inode. Therefore, `/x/y` and `/a/b` should point to the same file even after the file content is modified.

The current implementation of Prism supports hard links, but the description of the solution is beyond the scope of this paper. Zhao's thesis [20] contains the details of the solution.

### 3.3 Asynchronous Cloning

File sharing technique significantly reduces the cloning overhead, however, the Prism cloning mechanism can still incur nontrivial delay before the cloned filesystem is ready for use. The main reason is that the Prism cloning mechanism needs to traverse the parent filesystems and clone each encountered filesystem object individually, which incurs nontrivial overhead. On the other hand, to achieve selective cloning, Prism has to examine each filesystem object to determine whether the object should be excluded from cloning or not.

To better understand the impact of the filesystem traversal on the cloning performance, we conducted an experiment to clone a Fedora Core 4 system.

The filesystem contains around 170K files and 17K directories. The total size is around 6G bytes. Prism spent approximately 58 seconds to finish the cloning task. More than 70% of the cloning time is devoted to directory traversal. While this latency is acceptable in some scenarios, such as an administrator wishing to create new clones for distribution, it is not good enough for many applications such as testing untrusted applications. From the perspective of end users, they always hope to get a usable filesystem as quick as possible. Aiming at this goal, we developed an *asynchronous cloning* mechanism for Prism. For easy comparison, we call the cloning mechanism described in previous subsection *synchronous cloning*, because it blocks any requests to the cloned filesystem until the cloning procedure is completed.

The asynchronous cloning mechanism provides the same cloning semantics as the synchronous cloning mechanism, but is able to return a usable parent and cloned filesystem almost immediately (less than 1 second in all experiments we have conducted). It presents an illusion that the entire directory hierarchy is completely replicated, as in synchronous cloning, but the replication actually occurs in the background using a kernel thread. The background thread traverses the parent filesystem starting from the root to clone the directory tree, but also aggressively processes a file if it is accessed by the parent or the clone prior to the completion of the filesystem cloning. Eventually, the final state of the directory hierarchies in the fully cloned system is identical to that produced by synchronous cloning.

When a user requests to clone a filesystem, the asynchronous cloning mechanism usually replies to the user that the cloned filesystem is ready for use within a few seconds. A user will reasonably start to access either the clone or the parent filesystem. The asynchronous cloning mechanism must be carefully designed to present the same semantics to users as a fully copied filesystem. In particular, we must properly address the following two situations:

1. *A user in the cloned filesystem may access a file that has not been cloned.* Prism should quickly respond to the user, rather than blocking the user until the cloning thread eventually encounters and clones the file. As described earlier, the cloning thread works in the background to recursively traverse the parent filesystem and clone each encountered filesystem object. However, for a large filesystem, it can take a few minutes before the cloning thread encounters the requested file, which can be too long for the user to wait.
2. *A user in the parent filesystem can modify a file that has not been cloned.* Under such condition, Prism must ensure that the file is cloned before being modified. According to the cloning semantics, a clone should be identical to the parent filesystem's snapshot taken at the beginning of cloning procedure. Any modification to the parent filesystem afterwards should be transparent to the clone. However, if a file in the parent filesystem is modified before being cloned, the modified content will be exposed to the clone. Therefore, the asynchronous cloning mechanism must ensure that a file in the parent filesystem is cloned prior to modification.

Next, we describe how Prism handles the requests that are issued in the cloned or parent filesystem before the cloning procedure is finished.

**Handling Requests in the Cloned Filesystem.** When a user in the cloned filesystem issues a request to a file that has not been cloned, Prism aggressively clones the file on demand before processing the user's request. This avoids blocking the user too long.

The Prism on-demand cloning mechanism is based on two observations:

1. Before a process can access a file, Prism first *looks up* the file.
2. Before looking up a directory for a file, Prism must call the *permission* function to check that the requesting process has sufficient rights to access the directory.

Based on these two observations, Prism implements the on-demand cloning mechanism by extending the standard permission checking function.

First, we develop a core function, `pext3_expand_dir()`, that expands a directory at a time. Note that we use the term "expand" instead of "clone", because this function does not recursively go down a directory to clone all filesystem objects. Given a source directory, the `pext3_expand_dir` function only clones the filesystem objects that are directly under this directory. The function clones regular files as described in Section 3.2. However, for each subdirectory under the source directory, the function only creates an empty subdirectory at the corresponding location in the clone. In other words, this function only expands one level of directory hierarchy, and will not go deeper into subdirectories. The function flags each subdirectory as "UNEXPANDED" and associates it with the inode number of the corresponding source directory. To record this information, we add two fields, `i_expanding_flags` and `i_srcino`, to each pext3 inode. For a regular file, these two fields are not used. For a directory, however, these two fields indicate whether the directory is expanded or not. If all filesystem objects directly under a directory are cloned, the directory will be flagged as "EXPANDED". Note that a directory being flagged as "EXPANDED" does not mean that all its subdirectories are expanded. Normally, after running the `pext3_expand_dir()` on a specified directory, this directory is flagged as "EXPANDED", but all its subdirectories are still empty and flagged as "UNEXPANDED".

Next, as shown in Figure 5, Prism combines the core `pext3_expand_dir()` function with the standard permission function to perform asynchronous cloning. Given a directory, Prism first determines whether the directory is expanded or not by checking the directory's `i_expanding_flags` field. If the directory is expanded, Prism jumps to the original permission function. If the directory is not expanded, Prism calls the `pext3_expand_dir()` function to expand the directory, and then calls the original permission function.

With the Prism asynchronous cloning mechanism, a parent filesystem object is cloned by one of the two threads shown in Figure 5. The first thread is the *background cloning thread* that recursively traverses the parent filesystem and clones each encountered filesystem object. The second thread is an *on-demand*

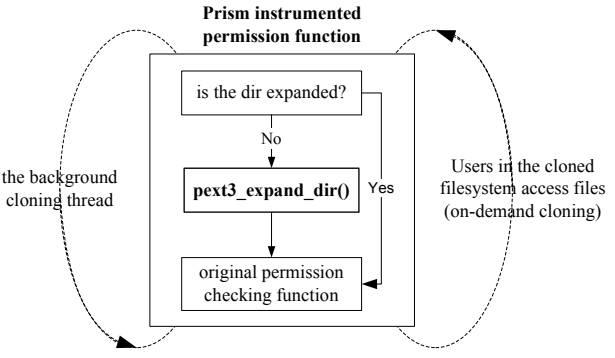


**Fig. 5.** Prism on-demand cloning

*cloning thread* that aggressively clones the filesystem objects that are requested by users in the clone.

We first discuss how the background cloning thread works. Upon receiving a request to clone a filesystem, Prism first creates the root directory in the clone. Next, Prism flags the directory as "UNEXPANDED" and associates it with the parent filesystem's root inode. Then, Prism would start the background cloning thread to clone the rest of the filesystem and then returns, presenting the user an illusion that the cloning task is completed immediately. The background cloning thread works as a directory walker that recursively traverses the entire cloned filesystem starting from the root directory. The background cloning thread looks up a file with an arbitrary filename in each encountered directory. The lookup operation is only used to trigger the permission checking function, which in turn expands the directory if it is not expanded. Thus, along with the background cloning procedure recursively traversing the cloned filesystem, the Prism permission checking procedure will be invoked to expand all encountered directories, which clones the parent filesystem in the background.

The on-demand cloning thread works in a similar way to aggressively clone the filesystem objects that are requested by users in the clone. When a user in the cloned filesystem accesses a file, Prism must look up the file before processing the request. This will trigger the permission checking function to expand all directories from the root to the parent directory of the file to be accessed.

To rapidly respond to end users' requests, Prism allow administrators to lower the priority of the background cloning thread with Linux command `nice`. As such, the background cloning thread will not contend with interactive sessions for disk bandwidth. Accordingly, the background cloning time could increase.

We use an example to illustrate the on-demand cloning procedure. Suppose a user accesses a file `/a/b/c`, but only the root directory `/` has been expanded. Prism first looks up the directory "/" for the entry "/a". The permission checking function is invoked to check the access permission of directory "/". Because the "/" directory has been expanded, Prism simply jumps to the normal permission checking procedure. Next, Prism looks up the directory "/a" for the entry

"/a/b". The permission function is invoked again to check the permission of "/a". At this time, the directory "/a" is not expanded yet and flagged as "UNEXPANDED". Prism then calls the `pext3_expand_dir()` function to expand the directory "/a". The directory entry "/a/b" is created but flagged as "UNEXPANDED". By repeating the above procedure, Prism expands the directories from "/" to "/a/b/". Eventually, when Prism looks up the file "/a/b/c", it has been cloned on demand. Note that the user process can run in parallel with the background cloning thread. With the common permission checking function, Prism seamlessly adjusts the cloning order and aggressively clone the directories needed for the file request, which achieves the on-demand cloning.

**Handling Requests in the Parent Filesystem.** With the asynchronous cloning mechanism, a user can get a command prompt before the parent filesystem is completely cloned. Accordingly, a user can write a file in the parent filesystem before the file is cloned. If the pext3 filesystem were to allow such an operation, it breaks the cloning semantics — the modification in the parent becomes visible to the clone.

One way to preserve the consistency of the parent filesystem is to clone from a snapshot of the parent filesystem. Many filesystems such as WAFL [2] and ZFS [7] provide the snapshot feature. We can adapt an existing mechanism to take a snapshot of the parent filesystem before starting the cloning procedure. This approach, however, requires substantial changes to disk and filesystem structure. As an alternative, Prism preserves the parent filesystem's consistency by detecting and aggressively resolving the consistency issues during background cloning.

Before starting the cloning procedure, Prism flushes the parent filesystem's dirty pages to disk, which eliminates the inconsistency caused by the buffered data. This procedure normally takes less than 1 second. During the period of cloning, Prism monitors the operations on the parent filesystem. If a process attempts to write a file in the parent filesystem that has not been cloned, Prism blocks the process, aggressively clones the file, and then resumes the process to write the file.

An important step in the above procedure is to tell whether the file to be changed is cloned or not. This step must be efficient, because it is critical to the filesystem performance. A naive approach to determine a file's cloning status is to maintain the list of files that have been cloned. By looking up the list, one can determine a file's cloning status. However, it would be slow to look up the file list if the filesystem is large and has a lot of files.

Another way to determine a file's cloning status would be to associate a flag with each parent file indicating whether the file has been cloned or not. However, this solution would require that Prism initialize the cloning status of each file in the parent filesystem before the cloning procedure is started. Otherwise, there would be no easy way to tell whether a specific file is cloned by current or previous cloning procedures. However, the initialization procedure would have taken substantial time for a large parent filesystem, significantly offsetting the benefit of the asynchronous cloning mechanism.

Prism addresses the above problem with three timestamps:

- *The global logical timestamp.* Prism maintains a global logical timestamp to record the occurrence time of cloning events. The logical timestamp is a 32-bit unsigned integer and is initialized to zero. This logical timestamp is incremented at the beginning of each cloning task.
- *The clonestart timestamp.* Prism maintains a `clonestart` timestamp for each filesystem to be cloned. The `clonestart` timestamp is normally equal to zero. When Prism starts to clone a filesystem, it sets the parent filesystem's `clonestart` timestamp to the current value of the global logical timestamp. When the entire cloning task is finished, Prism resets the filesystem's `clonestart` timestamp back to zero.
- *The lastclone timestamp.* Prism maintains a timestamp, called *lastclone timestamp*, for each filesystem object to record the last time when the object is cloned. The `lastclone` timestamp is stored in a 4-byte field, `lastclone`, in the directory entry of the filesystem object. Every time a filesystem object is cloned, Prism updates its `lastclone` timestamp to the current value of the parent filesystem's `clonestart` timestamp.

Prism is able to determine whether an original file has been cloned or not by comparing the parent filesystem's `clonestart` timestamp with the file's `lastclone` timestamp:

- *clonestart* will never be smaller than *lastclone* when the filesystem is being cloned. If a filesystem is not being cloned, its *clonestart* timestamp is 0. Prism can serve any operations to the parent filesystem under such condition.
- If $clonestart == lastclone$, the original file has been cloned by current cloning procedure.
- If $clonestart > lastclone$, the original file has not been cloned yet.

If a file in the parent filesystem is to be written, Prism first determines whether the file has been cloned or not with the above mechanism. If the file has been cloned, Prism can apply the modification to the file immediately without breaking the consistency of the clone. Otherwise, Prism must first resolve the consistency conflict before applying changes to the parent file. To do so, Prism blocks the writing process, aggressively clones the file, and then unblocks the writing process and serves the write request.

**Cloning Open Files.** Prism is designed to deliver a clear cloning semantics — the clone is identical to the parent filesystem's snapshot taken at the beginning of the cloning procedure. After a cloning procedure starts, all file modifications made to the parent filesystem are isolated from the cloned filesystems.

The current Prism implementation assumes that there are no open files in the parent filesystem when a cloning command is issued. Based on this assumption, Prism monitors the `open` requests to detect write operations in the parent filesystem. If a process requests to open a file for writing after the cloning procedure begins, Prism will regard the open request as a file modification operation. To

preserve cloning semantics, Prism blocks the writing process, aggressively clones the target file to the clone, then unblocks the writing process.

The above assumption could be too strong in real world scenarios. A file in the parent filesystem could be opened *before* the cloning procedure is started. Our current prototype does not address this scenario, but it can be addressed by aggressively cloning all open files before starting the background cloning thread. Upon receiving a clone command, Prism would first suspend the parent filesystem, clone the open files, and then reactivate the parent filesystem. Alternatively, we could have intercepted write operations and cloned at that point. We plan to evaluate these alternatives in the future.

Overall, asychronous cloning has the advantage that both the parent filesystem and the cloned filesystem are usable immediately even before the cloning task is finished. However, the access performance can be lower than normal if one attempts to access a file that is not cloned. Normally, the latency caused by file accesses during cloning should be small in practice because the set of files that are accessed during cloning is usually small compared to the whole filesystem.

### 3.4 Lazy Cloning

Prism provides another asynchronous cloning mode called *lazy cloning*. The lazy cloning mode is similar to the standard asychronous cloning mode, except that Prism does not start a background thread to clone the entire parent filesystem. All files are cloned on demand. In other words, it only clones a file when it is accessed.

The major advantage of this mode is that it only consumes little system disk and CPU resource. Prism does not need to pay any cost to clone the files that are never accessed. This is particularly useful for scenarios that only need a ephemeral filesystem. Software testing is a good example. Users often tend to destroy the clone after they test a untrusted application. It is often unnecessary to clone the entire filesystem for such an ephemeral system. The lazy cloning mode is also useful for evaluating the performance impact of the asychronous cloning mechanism on the cloned filesystem. It gives a worst-case bound of cloning penalty incurred by access to a cloned filesystem, because each accessed file is cloned on-the-fly.

However, we do not choose this cloning mode as the default Prism cloning mode. As discussed earlier, if a cloning job is complete, both the parent and cloned filesystems can be accessed as a normal filesystem without incurring additional cloning overhead. In contrast, before the parent filesystem is fully cloned, modifications to the parent filesystem can potentially cause consistency conflicts. When such conflicts are detected, Prism has to temporarily block the modification operations until the conflicts are resolved on-the-fly. The resolving latency would negatively impact end users' experience. To minimize the "impact window", one may want to finish the cloning as soon as possible. Therefore, we choose the standard asynchronous cloning as the default cloning mode in Prism.

### 3.5 The Prism Copy-on-Write Mechanism

In Prism, copy-on-write (CoW) must be performed if a VM writes a shared file. The CoW operation can be implemented as file copying. However, that can incur unncessary overhead, making operations like "`chmod`" inefficient. As an alternative, Prism employs a block-level CoW mechanism that is similar to Ext3Cow [10]. In Prism, each file is regarded as an inode associated with data blocks. Prism allows a file's inode and blocks to be shared separately. When performing CoW on a file to be changed, Prism only replicates the modified part, and still shares the unmodified part between the old and new copies. To track the reference counts of blocks, Prism deploys a reference count table for each block device. Each table entry is a one-byte reference count corresponding to a 4KB data block (default block size in pext3). A data block's reference count records how many *files* share the data block. If a block is shared by more than 255 files, it will be duplicated to a new block to avoid reference count overflow. We use the Linux journalling layer (*JBD*) to protect the block reference count table from being corrupted even if the system crashes in the middle of reference count updating.

### 3.6 Discussion

Prism uses hard links to achieve file sharing between the parent and cloned filesystem. Each hard link of a file will increase the file's reference count by one. In existing Unix-like systems, the maximum value of a reference count is 255. Therefore, if a same file is cloned for many times, the file's reference count can overflow. One solution is to make a physical copy when a file's reference count is about to overflow. This solution has not been implemented due to the time limit. While this solution incurs additional data copying overhead, we do not expect that it will substantially impact the Prism performance, because the reference count overflow issue is rare in a real world system. In addition, hard links are only entries in directory files. These entries are stored in each VM's own directory tree and will not affect other VMs' filesystem operation. Therefore, the increase of the number of hard links will not impact a VM's filesystem performance.

Prism's file sharing mechanism may incur security concerns. For example, one VM may attempt to modify a shared data block to disrupt other VMs. However, in Prism, a guest VM can only modify a file by issuing file system requests, which are subject to the Prism security checking. If a data block is shared by two or more VMs, copy-on-write operation will be performed to ensure the isolation between VMs.

## 4 Evaluation

Table 1 describes our evaluation platform. To facilitate a quick restoration of the operating system state to a consistent point for all experiments, we ran all the experiments in a DomainU Xen virtual machine, running a Fedora Core 4

**Table 1.** Experimental platform

| Hardware | |
|---|---|
| CPU | 3.00GHz Pentium IV |
| Memory | 512MB(Dom0) 512MB(DomU) |
| Disk | Maxtor 7200RPM EIDE |
| **Software** | |
| VMM | Xen 3.0.2 |
| Domain0 OS | Linux 2.6.16-xen0 |
| DomainU OS | Linux 2.6.16-xenU |
| Linux Distribution | Fedora Core 4 |
| Apache | version 2.0.58 |
| Connectathon | version 1.18 |
| Tar | version 1.15.1 |
| GNU gcc | version 4.0.2 |
| GNU ld | version 2.15.94.0.2.2 |
| GNU Autoconf | version 2.59 |
| GNU automake | version 1.9.5 |

distribution of Linux. The results reported are averages from multiple runs of the experiments. Generally, we found the results to be very consistent across the runs, with low standard deviation as compared to the average values.

### 4.1 Synchronous and Asynchronous Cloning Latency

We first evaluated the performance of the Prism synchronous and asynchronous cloning mechanisms. For the parent filesystem, we used a filesystem consisting of a Fedora Core 4 system with standard software packages, including around 170K files and over 17K directories. We cloned the filesystem with both mechanisms 32 times and reported the average time elapsed to clone the filesystem.

We first compared the cost of full copying versus synchronous cloning. The full copying of the filesystem took around 10.5 minutes (630 seconds), while synchronous cloning took 58.7 seconds. This clearly demonstrated that the Prism's file sharing technique significantly reduces the cloning overhead.

We then measured the time used by the Prism's asynchronous cloning mechanism to clone the filesystem. With the asynchronous cloning mechanism, the cloning activity largely occurred in the background. Prism instantly presented the users with an accessible filesystem clone. The observed latency was 0.18 seconds. The time spent by the background thread to clone the filesystem is about the same as that used by the synchronous cloning mechanism. This experiment shows that the asynchronous cloning mechanism significantly reduces the latency before the cloned filesystem is ready for use. This helps improve users' experience in filesystem cloning and makes it more practical to perform tasks such as testing untrusted applications in VM clones. While the asynchronous cloning mechanism hides the cloning latency from end users, it does not reduce the

cloning overhead. The total time used to clone the filesystem in the background was approximately the same as that of the synchronous cloning mechanism.

We also measured the disk space used by clones after each round of the cloning operation. The experiments showed that the used disk space consistently increased by 77MB each clone. This disk space is used to store a separate directory tree structure for each clone. The size of the fully copied filesystem is around 6GB. The clone size is around 1.3% of the disk space used by the fully copied filesystem before any modification to the clone. We expect that the disk space used by the clone will increase over time but will still be smaller than a fully copied filesystem, because the files that are never written can still be shared without duplication.

### 4.2 Performance on the Apache Workload

The asynchronous cloning mechanism presents end users a usable filesystem before the cloning procedure is finished. When a user in the cloned filesystem accesses a file that has not been cloned, Prism has to aggressively clone the file before processing the user's file request. Therefore, the asynchronously cloned filesystem could be slower than a fully cloned filesystem before the background cloning procedure is finished.

We used an Apache build task as a representative of typical workloads on a normal development machine to evaluate the performance impact of asynchronous cloning. In our experiment, Apache 2.0.58 was used as the benchmark object. The Apache archive includes 2339 files scattered in 188 directories. The total size of the archive is 6.13MB before being decompressed. After being decompressed, the total size of the Apache directory is 32.9MB. The benchmark first `unpacks` the archive of Apache 2.0.58 into a source directory. Next, it runs `configure` to build the source code dependency, which involves lots of small data read and file lookup operations. During the third phase, it `builds` the Apache binaries from the source files, which is a CPU intensive task, but also generates a lot of object files and temporary files. Finally, it `removes` all Apache files including the Apache source tree, generated configuration files, object files, and Apache executable binaries.

In practice, it is hard to consistently reproduce the dynamics when the benchmark and background cloning procedure run concurrently. The benchmark result can vary with different execution orders and time patterns. For this reason, we used the "lazy" cloning mode described in Section 3.4 — Prism only clones files on-demand and does not run the background thread to clone the unvisited files. As such, all files that are accessed by the benchmark will be cloned at runtime and all cloning penalties related to the benchmark are included into the benchmark result. This provides a stable evaluation on the performance penalty caused by the on-demand cloning mechanism. As another comparison point, to get the best-case performance for Prism, we also ran the benchmark on a fully cloned filesystem, which excludes the cloning overhead from the benchmark results. We compared the ext3, lazily-cloned, and fully-cloned pext3 filesystems on the Apache workload.

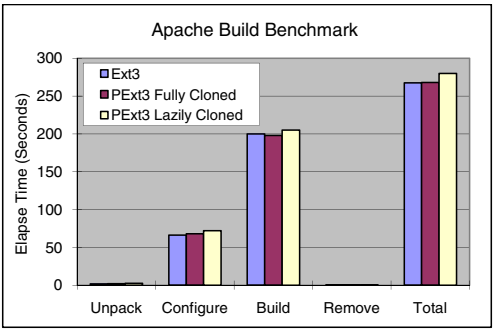


**Fig. 6.** Performance of Apache build workload. "Ext3" stands for standard Ext3 filesystem;"PExt3" stands for the Prism Ext3 filesystem.

Note that the benchmark needs to use some system tools and libraries such as `tar`, `gunzip`, and `gcc`. In our experiments, the benchmark process used the tools on the cloned filesystem. We guaranteed that by using "`chroot`" [6] to the cloned filesystem before running the benchmark. As a result, all input and output files needed for the benchmark are accessed from the cloned filesystem. To avoid warm cache effects caused by previous runs, we always ran the experiments right after the filesystem was mounted.

In Figure 6, each bar group shows a phase of the Apache build benchmark, while the "Total" group represents the total time consumed in the four phases of the benchmark. Overall, the Apache build benchmark running on a lazily cloned filesystem was 4.6% slower than on a full copied filesystem. With a fully cloned filesystem, the performance difference with `ext3` was negligible. These results demonstrate that the performance impact of the Prism asynchronous cloning mechanism is not significant. Moreover, once the background cloning procedure is finished, the cloned filesystem can be accessed at the same speed as a fully cloned filesystem.

### 4.3 Connectathon Test Suite

We used the Connectathon test suite [1] to evaluate operational correctness and performance of the pext3 system. The Connectathon test suite is a standard benchmark widely used by many filesystem projects such as Frangipani [15] and Ext3cow [10] to verify the correctness of filesystems and their interoperability with variety of operating systems.

We modified the Connectathon parameters to invoke more filesystem operations than the default setting. As such, we can better exercise the system and get more accurate performance results. In the experiment, we allocated a dedicated disk partition and used the pext3 and ext3 filesystems to manage this disk partition, respectively. For each setting, we ran the "basic" series of Connectathon benchmark for 10 times. The "basic" series of Connectathon test

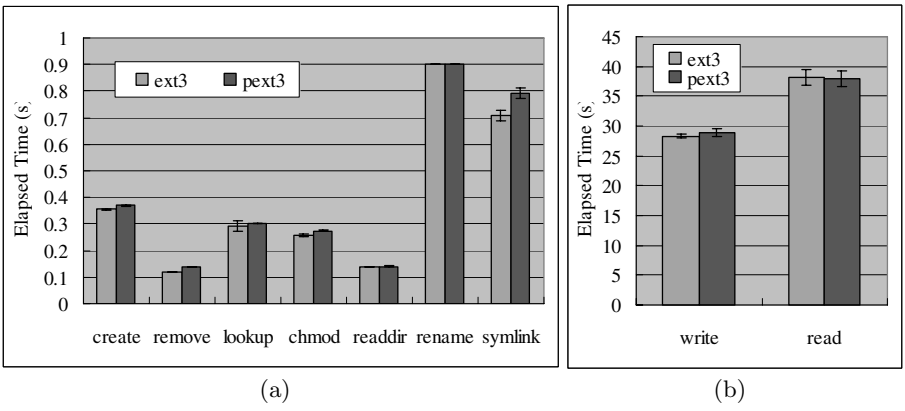


**Fig. 7.** Connectathon benchmark results. Each block group shows the latency of a step of the Connectathon benchmark.

includes nine steps. Each part tests a separate system call. In order, the nine steps are: (1) create 12400 files 62 directories 5 levels deep, (2) remove these files, (3) 20000 getcwd and stat calls, (4) 80000 chmods and stats, (5) create and write 1000 files. The size of each file is 1MB. Next, we read the 1000 files into 8K buffers sequentially. (6) create 400 files in a directory and read the directory entries for 81000 times using readdir, (7) create 200 files, rename and stat these files for 4000 times, (8) create 200 files, and perform symlinks and readlinks for 8000 times, and, lastly, (9) perform 15000 statfs calls.

The average time elapsed to run the benchmark on the pext3 and ext3 filesystems were measured and compared. The reported results are the average value from ten rounds of benchmark and reflect 95% confidence interval. To avoid warm cache effects caused by previous runs, we rebooted the test VM before each round of benchmark, and conducted the experiments right after the system is started.

The performance comparison is illustrated in Figure 7. Overall, the micro-benchmark results indicate that the pext3 filesystem delivers performance comparable to the ext3 filesystem on operations `create`, `lookup`, `chmod`, `readdir`, `rename`, `write`, and `read`. The performance differences between pext3 and ext3 on these operations are at most 6.23%.

Pext3 performs 16.67% and 11.27% slower than ext3 on the `remove` and `symlink` operations, respectively. For the ext3 filesystem, a `remove` operation mainly involves the updates on metadata including directory entries and block bitmaps, which are very efficient. The pext3 filesystem, however, uses the reference count table to track the usage status of data blocks. When removing a file, the filesystem driver must update the reference count (see Section 3.5) for each data blocks used by the file, incurring additional overhead. While the overhead of updating the reference count table is not substantial in term of the absolute value, it can be more pronounced for the filesystem operations that only incur

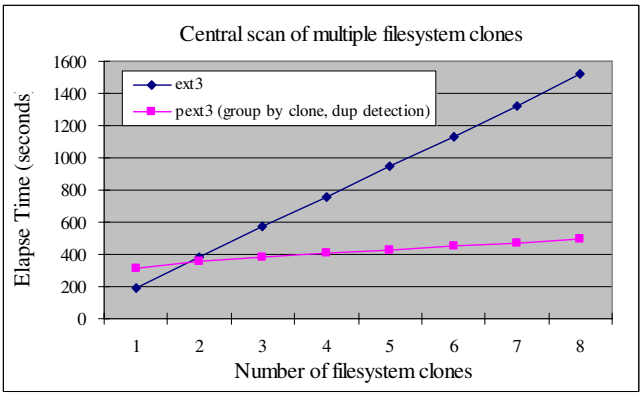


**Fig. 8.** Performance of central scanning 8 clones of a filesystem. "ext3" stands for fully copied filesystems. "pext3 (group by clone dup detection)" stands for cloned filesystem.

very low overhead. The same reason also explains the performance difference for the `symlink` operations.

### 4.4 Central Scan of Multiple Clones

Because of the way Prism does cloning (clones via file sharing), it is very easy for applications to identify files that are shared across filesystems. This allows faster versions of centralized applications that scan multiple filesystems (e.g., virus scanning or comparing two filesystems for changes) to be designed. The enhanced central applications can detect the files shared by multiple filesystems and scan them only once. We developed a central virus scanner that can check multiple filesystems (clones) for viruses. To see if Prism provides such performance advantages to the central scanner, we cloned the parent system 8 times. We then sequentially scanned $n$ cloned systems and compared the performance with scanning $n$ copied filesystems. Both the fully copied filesystem and the cloned filesystem have 170K files and 17K directories. The Prism central scanning tool maintained a list of scanned files' inodes (retrieved via the Linux `stat()` call) in a hash table. If it encounters the same inode again from another filesystem, it does not re-scan the file content.

Figure 8 shows the central scanning performance on both pext3 and ext3 filesystems. For $n = 1$, `ext3` outperformed `pext3` because every file had to be scanned in both systems. For small n, a cloned system is not expected to perform as well as a fully-copied system because it may have less spatial locality on the disk. Moreover, when scanning a Prism cloned filesystem, the central scanner needs to build up the hash table, which incurs additional overhead. For larger values of $n$, scanning cloned systems outperformed scanning copied systems by a significant factor. For `pext3`, there is still some increase in time with $n$ because the directory structure still has to be traversed $n$ times, but the slope is around 10 times lower.

## 5 Conclusion

This paper describes the design of a virtualized file system for supporting hosted servers and utility computing. Prism provides a file-level cloning mechanism that can clone any selected part of a VM's filesystem to a specified location. Prism is implemented by modifying ext3 with about 5000 lines of code. Prism uses an asynchronous cloning technique that establishes most of the file sharing in the background and also aggressively on demand. This technique allows both systems (parent and clone) to be usable almost immediately, irrespective of the size of the cloned filesystem. On the server side, Prism permits fast centralized scanning. Any files that have not been modified among parent and child filesystems have identical inode numbers and need to be only scanned once.

We implemented and evaluated the Prism cloning mechanism. The Prism cloning mechanism was able to clone a filesystem with around 170K files and 17K directories within 58.7 seconds, and return to the user an usable file system clone within 0.18 seconds. In contrast, copying the same file system takes more than 10 minutes. We also evaluated the performance of Prism's cloned file systems. On the Connectathon benchmark and the Apache build workload, a Prism cloned file system's performance is close to that of a standard ext3 file system. For applications that require scanning or comparing multiple cloned filesystems, Prism was found to be significantly faster.

## References

1. Connectathon. Introduction to the Connectathon NFS Testsuite (2007), `http://www.connectathon.org/nfstests.html`
2. Hitz, D., Lau, J., Malcolm, M.: File system design for an NFS file server appliance. In: WTEC 1994: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, Berkeley, CA, USA, p. 19. USENIX Association (1994)
3. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.: Scale and performance in a distributed file system. ACM Transactions on Computer Systems (TOCS) 6(1), 51–81 (1988)
4. Klivansky, M.: A thorough introduction to flexclone$^{TM}$volumes. Technical Report TR3347, Network Appliance Inc. (October 2004)
5. Liang, Z., Venkatakrishnan, V.N., Sekar, R.: Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In: ACSAC 2003: Proceedings of the 19th Annual Computer Security Applications Conference, pp. 182–191. IEEE Computer Society, Los Alamitos (2003)
6. McGrath, R.: Free Software Foundation. Chroot 5.2.1 - run command or interactive shell with special root directory, The Linux Manual Pages (May 2005)
7. Sun Microsystems. Solaris ZFS - The Most Advanced File System on the Planet (2007), `http://www.sun.com/software/solaris/ds/zfs.jsp`
8. Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S., Smith, F.D.: Andrew: a distributed personal computing environment. Communications of the ACM 29(3), 184–201 (1986)

9. Muniswamy-Reddy, K., Wright, C.P., Himmer, A., Zadok, E.: A Versatile and User-Oriented Versioning File System. In: Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004), San Francisco, CA, pp. 115–128 (2004)
10. Peterson, Z., Burns, R.: Ext3cow: A time-shifting file system for regulatory compliance. ACM Transcations on Storage 1(2), 190–212 (2005)
11. Pfaff, B., Garfinkel, T., Rosenblum, M.: Virtualization aware file systems: Getting beyond the limitations of virtual disks. In: NSDI 2006: Proceedings of the 3rd Symposium of Networked Systems Design and Implementation, pp. 353–366 (May 2006)
12. Satyanarayanan, M.: Scalable, secure, and highly available distributed file access. Computer 23(5), 9–18, 20–21 (1990)
13. Soules, C.A.N., Goodson, G.R., Strunk, J.D., Ganger, G.R.: Metadata efficiency in versioning file systems. In: FAST 2003: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, pp. 43–58. USENIX Association (2003)
14. Sun, W., Liang, Z., Venkatakrishnan, V.N., Sekar, R.: One-Way Isolation: An Effective Approach for Realizing Safe Execution Environments. In: NDSS 2005: Proceedings of the Network and Distributed System Security Symposium (2005)
15. Thekkath, C.A., Mann, T., Lee, E.K.: Frangipani: a scalable distributed file system. ACM SIGOPS Operating Systems Review 31(5), 224–237 (1997)
16. VMware. VMware VMFS: High-performance cluster file system for storage virtualization (October 2006), `http://www.vmware.com/pdf/vmfs_datasheet.pdf`
17. Warfield, A., Ross, R., Fraser, K., Limpach, C., Hand, S.: Parallax: Managing storage for a million machines. In: Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS X), Santa Fe, NM (June 2005)
18. Yu, Y., Guo, F., Nanda, S., Lam, L.c., Chiueh, T.c.: A feather-weight virtual machine for windows applications. In: VEE 2006: Proceedings of the second international conference on Virtual execution environments, pp. 24–34. ACM Press, New York (2006)
19. Zadok, E., Iyer, R., Joukov, N., Sivathanu, G., Wright, C.P.: On incremental file system development. ACM Transactions on Storage (TOS) 2(3) (accepted) (August 2006)
20. Zhao, X.: Improving the storage manageability, flexibility, and security in virtual machine systems, Ph.D thesis, EECS Department, University of Michigan, Ann Arbor (2007), `http://portal.acm.org/citation.cfm?id=1368534&coll=GUIDE&dl=GUIDE`
21. Zhao, X., Borders, K., Prakash, A.: Towards protecting sensitive files in a compromised system. In: SISW 2005: Proceedings of the Third IEEE International Security in Storage Workshop, Washington, DC, USA, pp. 21–28. IEEE Computer Society, Los Alamitos (2005)