Prism: Lightweight Filesystem Virtualization Via Selective Cloning

Xin Zhao Atul Prakash Kevin Borders
University of Michigan, 2260 Hayward Street, Ann Arbor, MI, 48109-2121, USA
{zhaoxin,aprakash,kborders}@eecs.umich.edu

Abstract

Suppose filesystems supported cloning of any subset of directories, even those containing gigabytes of data, almost instantaneously, while guaranteeing isolation between the original copy and the cloned copy. We show that this simple abstraction can greatly simplify many routine tasks, both for a single operating system and for clusters of similar virtual machines in a virtual server farm. In this paper, we describe the design of Prism, a file server that supports lightweight cloning and centralized file management of multiple filesystems. Unlike copying in standard file systems or cloning operations on virtual disks, the Prism filesystem supports high-level operations to create fast clones of any part of a filesystem. After cloning, each cloned file system can evolve independently, providing similar read/write semantics as if the file systems were copied from each other. This abstraction is useful for supporting virtual server farms. It is possible to create filesystems for new virtual machines interactively, while protecting private home directories, log files, and password files. Furthermore, when cloned systems have a high degree of similarity, Prism makes centralized scanning operations (e.g., for malware checking) for multiple cloned file systems significantly faster than scanning each file system individually. Prism's capabilities are also applicable to individual operating systems. Prism can help provide lightweight file system isolation between processes to prevent untrusted code from damaging the filesystem.

Prism is currently implemented by extending the ext3 filesystem. On the Postmark benchmark and the Apache build workload, Prism's performance is comparable to that of a standard ext3 filesystem. For applications that require scanning or comparing multiple, cloned filesystems, Prism is significantly faster. This paper describes the design and evaluation of the Prism filesystem.

1 INTRODUCTION

Many storage systems provides a powerful feature that allows them to create clones of a data storage. Here, data storage refers to both block-level storages such as disk partition and volumes, and file-level storage such as file systems. A clone is semantically similar to a copy of the parent storage. Once created, it is independent of the parent storage. Subsequent changes to either one are not reflected in the other.

Conceptually, the cloning feature is great for any situation where testing or development occur, any situation where progress is made by locking in incremental improvements, and any situation where there is a desire to share data in changeable form without endangering the integrity of the original. We argue that a successful cloning mechanism must have two characteristics: *efficiency* and *flexibility*.

By efficiency, we mean that the speed of creating clones must be fast. For example, imagine a situation where a user wishes to test some downloaded software, but is not willing to risk damaging his or her filesystem. The user could create a clone of the filesystem and test the software on the cloned copy. If the cloning is

immediate, the testing process can be repeated rapidly from clean copies. The same task could be done by copying the filesystem, but this incurs substantial disk I/O operations and would probably be very slow, taking minutes to hours. The time that users spend waiting for such tasks can significantly change users' experience.

By flexibility, we mean a user should be able to selectively choose appropriate parts of the data storage for cloning. In many scenarios, the ability to exclude certain files or directories from cloning is very useful. For example, a user, say Ann, may have a working version of an operating system and applications (with her filesystem mounted from the Prism server). Bob wants to use the same OS configuration, and Ann is willing to allow Bob to copy her filesystem. But Ann wishes to exclude certain directories, such as her home directory, `/tmp` directory, and `/var/log` directory, for privacy reasons. It is desirable for the cloning mechanism to be able to exclude certain files or directories from cloning. Similarly, the ability of cloning only a part of a filesystem is also useful. For example, with this capability, a user can only clone her home directory at every login. As such, if her files are accidentally corrupted during the session, the user can restore them from the clone. Because the rest directories are irrelevant to this purpose, they are not cloned. This minimizes the performance impact caused by the cloning operation.

The cloning support can be implemented in various ways. The simplest solution is to copying data from the parent storage to create the clone. For a file system, it is easy to specify the files to be copied, which meets the requirement of flexibility. However, data copying incurs substantial disk I/O operations. It can take minutes to hours to copy a file system, making this solution very inefficient. Some current virtual machine systems, such as VMWare [25], support block-level cloning of virtual disks. They use a copy-on-write technique to avoid duplicating disk blocks that are the same in a clone as in the parent virtual disk. Only when shared blocks are modified do these systems create new copies of these blocks. The block-level cloning mechanism is fast and guarantees isolation. However, it falls short in flexibility. The cloning mechanism operates at block level and has no knowledge of file system semantics. This makes it difficult to selectively clone part of data storage, virtual disks in this case.

In this paper, we describe the *Prism* file system which provides a file-level cloning mechanism. The file-level cloning mechanism is similar to a copy operation in a standard file system, except much faster. From an end-user perspective, cloning is completed almost instantaneously, irrespective of the size of the cloned filesystem's size (either in number of files, depth, or total number of bytes). The filesystem clone is immediately ready for use without interrupting the access to the parent filesystem. Moreover, Prism allows a user to clone any part of a filesystem, while excluding the rest files and directories.

In addition, Prism allows multiple domains to share identical parts of a single filesystem copy, while keeping their modifications to the filesystem private to themselves. As such, Prism serves multiple domains with a single filesystem copy, while preserving the illusion that each domain has a dedicated file system copy. In contrast, most existing systems can either allow multiple domains to share a single file system copy without isolation or have to maintain a copy for each domain to achieve isolation between domains.

Prism helps improve the efficiency of central operations on multiple cloned filesystems. In the example with Ann and Bob, a system administrator may wish to periodically scan Ann's and Bob's filesystems for viruses. With Prism, it is straightforward to write a scanning application that scans common, unmodified files between the two systems only once. Similarly, comparing two filesystems, like `diff -r` on Unix, can be made much faster in Prism than on copied filesystems. This would be useful in the case of untrusted software testing scenario to determine if it has maliciously modified the filesystem.

Prism utilizes a copy-on-write technique that operates at both the file-level and the block-level to make cloning very fast. Though the copy-on-write principle is simple, several complications arise when attempting to make it fast for cloning large filesystems. In particular, our goal was to avoid traversing the filesystem hierarchy before being able to respond to the user. At the same time, we wanted the semantics of cloning to be identical to that of a copy at the time of the cloning operation. We explored several options for implementing selective cloning. The first option was *static cloning*. It uses a copy-on-write technique to avoid duplicating file data by making file clones share data blocks with the parent files. The static cloning mechanism presents end users a usable file system clone after it establishes data sharing for all files. Though this was considerably faster than a full copy of an entire filesystem, it still took approximately 104 seconds

for cloning the Fedora Core 4 distribution in an experiment conducted by us. We then improved on the scheme and implemented a *dynamic cloning* method in which most of the data sharing is established at the background and lazily on an as-needed basis. It provides the same semantic guarantees as static cloning, but allows both filesystems to be usable immediately. The Prism cloning mechanism uses disk space very efficiently. The parent and cloned filesystem share data of unchanged files, which usually occupy a large portion of files. Prism only consumes additional space to store changes between the parent and clone. This is a huge potential saving in dollars, space, and energy. In addition to all these benefits, clone have the similar performance as a fully copied clone.

We measured performance of a Prism-based system on several workloads and compared it to solutions based on the ext3 filesystem. On both the Postmark benchmark and an Apache-build workload on a cloned filesystem, the performance was comparable to that of an ext3-based filesystem, with only a minor performance penalty. The cloning operation itself was essentially an immediate operation from the perspective of the end-user, irrespective of the size of the filesystem, taking only 0.18 seconds to complete. For scanning multiple cloned filesystems, Prism outperformed an ext3-based solution significantly because it was able to skip over files that had not been modified since cloning. The performance advantage of Prism over ext3 went up as the number of clones was increased. We also implemented a `smartdiff` program to compare a filesystem and its clone that worked like a recursive diff program to compare two directories. The `smartdiff` program outperformed a standard recursive diff program by more than an order of magnitude when the two filesystems being compared shared many common files. In terms of disk space, a clone of a Fedora Core 4 distribution, consisting of over 170K files, took up about 1.3% of the space (77MB for the clone versus 6GB for the parent filesystem).

The rest of the paper is organized as follows. Section 2 discusses the advantages of the interactive cloning abstraction provided by Prism from an end-user perspective. It gives several applications that benefit from the availability of fast selective cloning. Section 3 describes the overall architecture and compoents of the system. Section 4 outlines the design of Prism, including issues in making cloning operation usable interactively. Section 5 presents performance results. Section 6 talks about related work. Section 7 concludes.

2 SELECTIVE CLONING APPLICATIONS

The selective cloning operation in Prism is conceptually simple. It provides similar semantics to creating a new copy of a specified part of the filesystem. The crucial difference is that cloning operation is efficient, even when the filesystem is large. Prism can work within a single operating system to allow cloning of any directory. However, it is also designed to provide a central storage for multiple virtual machines. Cloning can be used to rapidly instantiate new, customized filesystems from existing ones for new virtual machines.

This section presents several applications for use of Prism's selective cloning support and show how, in this settings, Prism offers same or better service than existing mechanisms such as virtual disks and network file systems.

2.1 Selective Cloning to Enhance Security

Consider the following scenario. A developer, Ann, is using a virtual machine that has all the right software installed. Another developer, Bob, wishes to avoid wasting time reinstalling the same software on a fresh virtual machine. He asks Ann to provide a clone of her virtual machine's filesystem. Ann would be happy to do that but is concerned about the privacy of her files. Prism's selective clone capability would allow Ann to make a clone of her virtual disk, while excluding specified files or directories from the clone. In Prism, the above task can be done from the administrative console very simply as:

```
% prism clonefs /annfs /bobfs -exclude /annfs/home/ann
```

Here, `/annfs` refers to the file system as seen by Ann's virtual machine (Ann in her VM will see that as her root file system). That file system is cloned into a new file system `/bobfs`. Ann's home directory

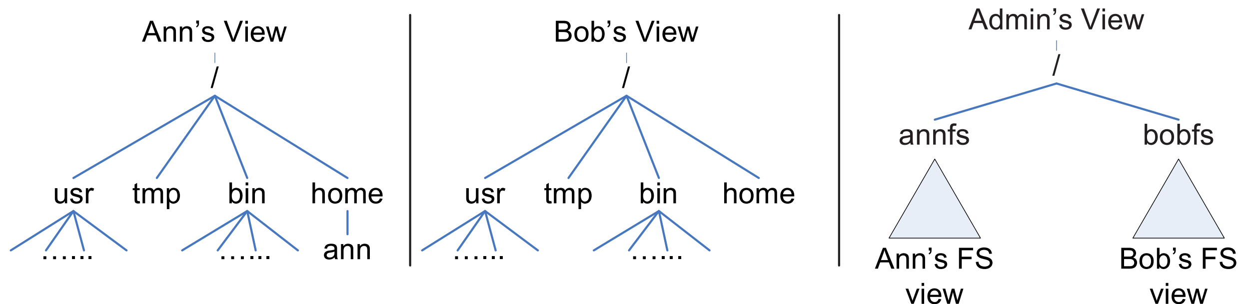

Figure 1: Prism File System Views

is excluded from cloning. Figure 1 shows the view of the file systems as seen by Ann, Bob, and by an administrator on the console at the Prism server. The entire operation completes interactively, typically in milliseconds, irrespective of the size of Ann's file system. Both file systems are usable immediately after cloning (the command usually returns back to the user in less than a second).

A more general version of `clonefs` allows the use of a template file as an argument. The template file specifies a directory tree for cloning, but can list the set of files/directories that should be excluded from cloning.

Let's consider a few alternatives for accomplishing this task using standard solutions available today. Ann could create a new filesystem partition for Bob, copy all the files that she is fine with sharing to the Bob's partition, and then provide the new partition to Bob's virtual machine. This works, but is obviously a slow process, which can take minutes to hours. We believe that the huge difference in elapsed time changes the user's experience substantially.

Another alternative would be for Ann to partition her filesystem *a priori* so that her private files and non-private files are never on the same partition. If that is done, then the entire non-private partition could be cloned using virtual disk cloning, as supported in VMWare and Xen. Prism does not require such a decision to be made in advance and is thus more flexible. In fact, default partitions for most operating systems, such as Fedora and Windows, do not create partitions with privacy as the deciding factor. Private content can also be spread out in multiple places, such as `/tmp` and in `log` files, which often do not reside on dedicated partitions. Prism requires no *a priori* decision to be made, separating the concerns of privacy (semantic decision) from that of physical disk partitioning (usually driven by performance, reliability, and backup considerations).

It is also possible for Ann to export certain directories for sharing with Bob via network file systems. However, she still faces a dilemma. If the exported directories is shared in read-write mode, Ann's VM can be disrupted if Bob's VM is compromised and the hacker change the executable files shared between Ann and Bob. If the exported directory is shared in read-only mode, Ann must carefully relocate the files that must be modified at runtime out of the directories to be shared. Otherwise, Bob's VM cannot run normally. In practice, such kind of files can scatter around the entire file system of Ann, making the relocation task nontrivial.

2.2 Efficient Scans of Multiple Filesystems

Scanning a filesystem is a common operation for virus/spyware checking, building an initial search index on a file system (e.g., as done by Google Desktop) and for file system backups.

Suppose a computer need to run multiple VMs whose software environments are similar with minor difference. This is not a rare scenario in real world. For example, a website can run three VMs on a single computer to implement the 3-tier web architecture: one VM as the frontend to process requests for static webpages, one VM as application layer to run CGI programs and process dynamic web requests, and one VM to provide services such as database. The NetTop system developed by the national security agency

(NSA) is another example. One can deploy multiple VMs with identical software environment but different privileges so that applications with different security levels can run on a single computer.

Prism helps substantially reduce the time and overhead of deploying such kind of systems. A system administrator can use the first VM's software environment as a template to create multiple clones and then customize them to meet different requirements.

With Prism, it is possible to scan above systems centrally for tasks such as malware detection, rather than relying on each virtual machine (which is under user control) to do the scans. Central scanning has two major advantages over scanning within each VM. First, the central scanner is independent of any other VMs and cannot be compromised or bypassed, which is critical for virus checking. In contrast, virtual disks provide no file-level knowledge. The central scanner has to understand all file systems that are deployed on virtual disks in order to scan them centrally, making central scanning less practical. Virus scanners have to run within each virtual machines and are more vulnerable to attacks.

Second, Prism can help central scanners to detect files that are shared by multiple clones and scan them only once. This can significantly reduce the overhead of virus checking on multiple VMs. In contrast, with virtual disks, each VM must run virus scanner in its guest OS. A virus scanner can only access to the virtual disks of its own VM. Therefore, it cannot detect the data blocks shared by multiple VMs. The virus scanner may have to read a same block multiple times, making the scanning procedure less efficient. Our performance experiments show significant improvements over independent scans on each file system.

Network file systems can support central scanning as well. But as Section 2.1 shows, due to the lack of isolation, VMs can be even more vulnerable to attacks if a sharing VM is malicious.

If a configuration file, such `/etc/passwd`, is cloned, then it is possible to take advantage of cloning to modify it across all systems by changing a single copy on disk. We do not discuss this type of modification in this paper as it breaks the standard isolation guarantees that Prism aims to provide. However, to assess the feasibility, we have implemented an experimental administrative mechanism to update a file clone without breaking the linkage between the clones. This may be a useful capability in the future if it is restricted to central administrators who are managing multiple file systems. Users on individual systems (including root users) still see a dedicated file system, except that the central administration is permitted to update the files on their systems. We do not discuss this mechanism further in this paper and assume that writes on a cloned filesystem are not shared with other clones.

2.3 Simplify IT operations

With Prism, it is possible to maintain multiple copies of production systems: live, development, test, reporting, etc. This feature is useful to simplify IT operations. For example, imagine a situation where the IT staff needs to make substantive changes to a production environment. The cost and risk of a mistake are too high to do it on the production volume. Ideally, there would be an instant writable copy of the production system available at minimal cost in terms of storage and service interruptions. By using Prism, the IT staff gets just that: an instant point-in-time clone of the production data that is created transparently and uses only enough space to hold the desired changes. They can then try out their upgrades using the Prism clone. At every point that they make solid progress, they clone their working Prism clone to lock in the successes. At any point where they get stuck, they just destroy the testing clone and go back to the point of their last success. When everything is finally working just the way they like, they can split off the clone to replace their current production system. The Prism's cloning feature allows them to make the necessary changes to their infrastructure without worrying about crashing their production systems or making untested changes on the system under tight maintenance window deadlines. The results are less risk, less stress, and higher levels of service for the IT customers.

2.4 Lightweight Snapshots

While Prism is not a versioning filesystem, it can be used to provide some of the benefits of a versioning filesystem because clones behave like filesystem snapshots. In Prism, it is straightforward to provide a

command to the user, allowing him to create a read-only clone of the home directory at every login, on demand, or periodically.

Another possibility is to modify programs like `login/sshd` to make a cloned copy of the file system at every login. If subsequent malicious activity is suspected during a login session, then one can compare the snapshots before the session and a snapshot after the session to determine which files were modified. This comparison operation can be implemented much more efficiently as a user application in the Prism-based filesystem because files that have not been modified at all will not require a content-comparison. More important, the comparison provides human-readable information. It shows the different files instead of blocks, making it easier for administrator to better use snapshots.

2.5 Executing Untrusted Code Within the Same OS

Consider a scenario where a professor wishes to execute the class projects of his or her students in the computer security class for grading but is worried about side-effects. It is possible that a buggy or malicious project could corrupt files on the system. With Prism, he or she could do the following:

`% prism clonefs / /gradecopy % chroot /gradecopy % compile/execute the project`

Finally, from a shell that is outside the root jail, the results can be copied back to the parent filesystem and the clone removed.

`% prism rmclone /gradecopy`

(This process can be repeated for other student projects.)

The student's project will be able to access to the entire file system in the chroot environment, including shared libraries, header files, etc, but will not affect the original file system as it will be working off of a cloned copy.

The alternative method for guaranteeing a similar level of isolation within an operating system today is to create separate user accounts for each grading session because the process isolation alone does not guarantee isolation of file systems. Note that using a single account is not adequate because of potential side-effects from one project on another. Without Prism's cloning primitive, the permissions on relevant files in the system would have to be carefully configured to guarantee that a userid used for the grading cannot tamper with those files on the system. Prism provides a guarantee that the original filesystem remains intact even if the filesystem permissions are configured incorrectly (as long as the root jail is secure).

For multi-layered security, cloning can be used in conjunction with running untrusted code with a restricted-right userid. In that case, untrusted code has to break both the root jail as well as exploit controls placed on the userid. For even further isolation guarantee, the untrusted code can be run in a guest virtual machine created with a cloned filesystem. In that case, the untrusted code can even be given root privileges on the guest VM.

2.6 Software Testing and Determining File System Impact

Consider the case where a user wishes to install third-party code on her system, but is not sure which files the executable will modify. With fast cloning, it is possible to install the code on a cloned file system. With Prism, one can then compute the difference of the original file system and its clone more efficiently than comparing a file system and its copy as shown below:

```
% prism clonefs <parentfs> <childfs>
% chroot <childfs>
% install third-party software.
```

Then, from a different shell, run

```
% smartdiff  -r <parenfs> <childfs>
```

The `smartdiff` is an application that we wrote to efficiently compute differences between a parent filesystem and a cloned filesystem (or any two filesystems that are both managed by a single Prism server). It works like a normal `diff`, with the exception of performance optimizations. It does not need to examine the contents of files that have not been modified (usually, a vast majority of files will fall in this category)

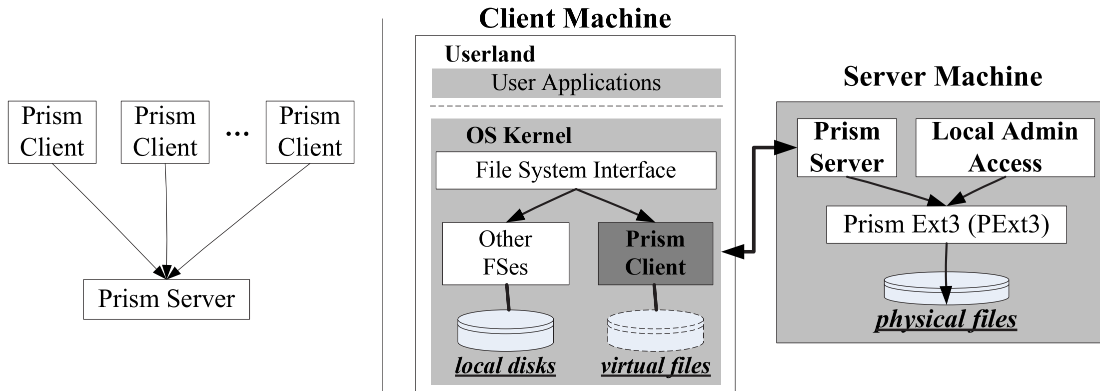

Figure 2: Prism File System Overview

because they will have the same inode value in both the parent and child filesystems. It also checks that `childfs` is not mounted as a subdirectory of `parentfs`, in which case it skips over `childfs` when traversing the directory tree of `parentfs` (and conversely). As a result, the following can be handled by `smartdiff`:

```
% prism clonefs / /testfs % smartdiff -r / /testfs
```

3 PRISM ARCHITECTURE OVERVIEW

As shown in Figure 2, a Prism system usually consists of one or more *Prism clients* accessing a common *Prism server*. The clients and the server could reside on the same physical machine (e.g., each client in a different virtual machine) or on different physical machines. At present, Prism only provides data storage for virtual machines running in a single computer.

3.1 Prism Server

The Prism server uses the Prism ext3 file system (pext3) to manage physical files. The `pext3` filesystem shown in Figure 2 is the `ext3` filesystem extended with Prism's cloning support. The Prism server is able to export one or more pext3 directories to remote clients using the NFSv3 protocol [15]. There is no technical difficulties of using other network file system protocols like Samba [24]. A client's Prism mounted filesystem is essentially a pext3 directory at the server side.

The pext3 file system can clone any specified directory and export the clone as a file system to remote clients. When creating a clone, the `clonefs` command takes a template file as its argument. The template file specifies the parent directories to be cloned, the destination directory where the clones reside, and the directories that should be excluded from cloning. Only a Prism directory can be cloned. All clones are managed by the same Prism server, though they can be accessed by multiple clients. Once a filesystem clone is created, it can be accessed as a normal filesystem and supports all file operations such as read, write, as well as subsequent clone operations.

The pext3 file system is implemented as a kernel module. It does not change any system call interfaces and can be access by existing applications without any modification. The cloning operation, however, is unique to pext3 and cannot be invoked by standard file system calls. We therefore implemented a cloning tool that invokes the cloning operation by talking to the `pext3` kernel module via an `ioctl` channel. This is a fairly standard alternative to adding a new system call.

3.2 Prism Client

The Prism clients present applications a virtual file system, through which guest applications can read or write files, as if the files reside on a dedicated file system. The transparency is achieved with modest

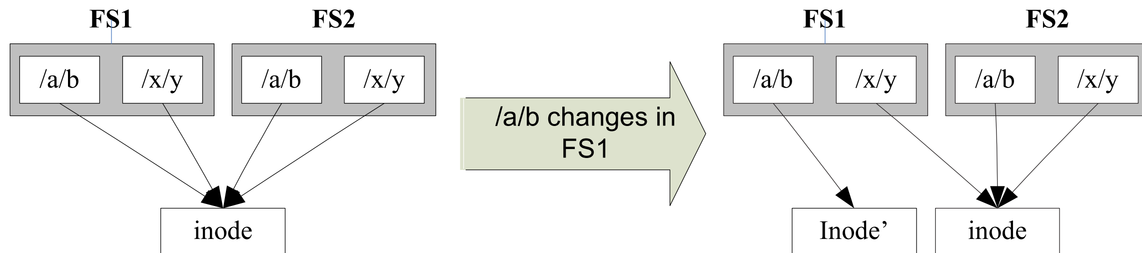

Figure 3: Copy-on-write on an inode breaks hard links within a file system. To preserve hard link semantics within FS1, both /a/b and /x/y should point to the new inode's inode.

change to the guest operating system – a kernel module must be inserted into the guest operating system. No modification is required for guest applications. The property of transparency causes the least incompatibility with existing applications and procedures.

To transparently present application a virtual file system, a Prism client registers itself with the common file system interface of the operating system to become a Prism file system handler. Most mainstream operating systems offer such kind of common file system interface to plug in the drivers of new file systems. For example, Linux systems use the *virtual file system (VFS)* interface [2], and Windows uses the *Installable File System Kit (IFS Kit)* [4]. At present, Prism is only implemented on the Linux platform. But the Prism design can be applied to the Windows platform. For the rest of this thesis, we use the Linux as an example platform to describe the design of Prism. For brevity of description, we use the Linux term "VFS" to refer to the "common file system interface" in the Prism model.

A Prism client presents to guest applications a "virtual" file system whose physical data resides in the Prism server rather in the local disks. An application accesses virtual files using standard system calls, such as `open`, `read`, and `write`. When the system calls get passed down to the kernel, the VFS layer determines whether the requested files reside within Prism and forwards the requests to the Prism client if necessary. The Prism client then transfers the requests to the Prism server and waits for the server response. After the client receives the server's reply, it returns the results to the guest application, giving the calling applications an illusion that the requests were processed locally.

4 Prism Cloning Mechanism

The cloning primitive has existed in block-level storage systems for many years. For example, VMWare [23] and Xen [1] use a block-level copy-on-write technique efficiently clone virtual disks. The block-level copy-on-write technique is simple but very efficient if the entire disk needs to be cloned. However, this block-level cloning mechanism cannot selectively clone parts of a storage (disk partition, volumn, or filesystem). To do this, one has to examine the directories/files structure, which is not exposed by block-level storage systems. To meet Prism's flexibility requirement, we must devise a file-level cloning mechanism.

4.1 Cloning Via File Sharing

The simplest way to clone a directory in a source filesystem is to copy it (`cp -r srcdir destdir`). However, this generates considerable disk I/O and consumes a lot of disk space, making many of the applications we considered in Section 2 less impractical with this solution.

The key idea in implementing efficient cloning is to avoid duplicating file data during cloning as much as possible. Based on this idea, Prism employs a copy-on-write technique to avoid copying files that are the same in a clone as in the parent directory.

Basically, Prism clone directories and *regular* (non-directory) files in different ways. When cloning a file system, Prism always starts from the parent file system's root directory to traverse the entire directory structure and clone all encountered file objects. For each directory, Prism creates a new directory at the

corresponding place in the clone. For a regular file, Prism creates a clone by hard linking to that file in the clone. In the pext3 file system, all named files are hard links. The name associated with the file is simply a label that refers the operating system to the actual data. As such, more than one name can be associated with the same data. The clone and the original file can therefore share the file data. 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 file system and its clone is preserved.

While the basic idea is conceptually simple, two challenges must be addressed in designing the selective cloning mechanism:

1. How to ensure that a clone provides the same semantics as a fully copied file system? As we will see later, cloning via file sharing can change the file access semantics. It is important to avoid such issue.

2. How to ensure that a clone has similar performance as a normal file system? In Prism, a clone shares files with its parent. If one attempts to modify a shared file, copy-on-write must be performed. However, it is nontrivial to minimize the performance impact of the copy-on-write operation.

4.1.1 Preserving File System Semantics

As we point out earlier, Prism clones regular files using hard links. It does not need to create a new inode for the cloned file and thus minimize the disk I/O. However, this can change the file system semantics if the parent file system has hard links.

Figure 3 shows an example that helps illustrate this issue. Suppose a user clones a file system `FS1` to a new filesystem `FS2`. Note that a exportable file system in pext3 is essentially a pext3 directory. 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. As illustrated in Figure 3, Prism creates two hard links in `FS2` for `/a/b` and `/x/y`, respectively.

Now suppose the file `/a/b` is modified in the parent filesystem. According to the standard UNIX file system 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, one should be able to see new data by reading the file `/x/y` in `FS1`. On the other hand, according to the cloning semantics, any modification made in one file system should be transparent to the other file system. The files `/a/b` and `/x/y` in `FS2` should still contain old data.

In order to preserve isolation between `FS1` and `FS2`, upon modification to file `/a/b`, Prism must perform a copy-on-write to create a new inode and data block to apply changes. However, the files `/a/b` and `/x/y` in `FS1` point to different inodes and contain different contents, which breaks the semantics of hard links. To address this problem, Prism must make the hard link `/x/y` point to the new inode as well. However, given an inode, there is no efficient way to find all hard links pointing to this inode. Prism must traverse the FS1's entire directory structure to find these hard links, making the file modification very slow.

To determine whether hard links are a real issue, we analyzed one of the Linux installations at a departmental server, and found that over 6000 files had multiple hard links to them. Other file systems, such as NTFS, also support hard links. We therefore concluded that it is desirable to maintain support for them if at all possible, since some applications and users clearly rely on them.

Our solution to this problem is based on the following observations. There are over 170K files on our experimental machine. Even though around 6K files were found to have multiple hard links, they only occupy a small proportion of the entire file system. We assume that the files with multiple hard links are relatively a small fraction of the total number of files in most file systems.

Based on this assumption, we decided to use a different way to clone regular files with multiple hard links. For simplicity of description, we call an inode's hard links in a local file system as *internal links*, and call its hard links in other local file systems as *external links*. When cloning a filesystem, if an inode has multiple internal links, we replicate it so that the clone has a private inode. Next, we insert the original inode number and the new inode number into a radix tree for fast searching. During the following cloning procedure, if a different file has the same inode number, we will create a hard link pointing to the corresponding new inode.

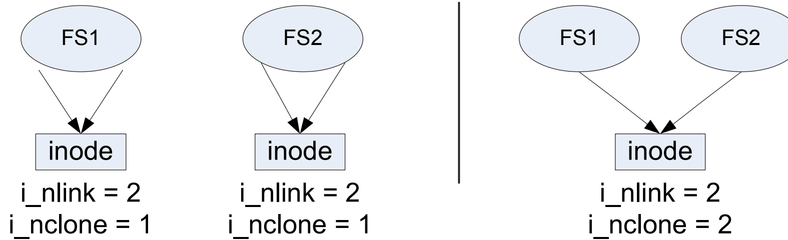

Figure 4: Clone of regular files

We impose the following restriction on the cloning procedure:

> An inode that has more than one internal links in a filesystem cannot be shared by another filesystem via external hard links (it must be replicated).

According to this restriction, Prism clones regular files as shown in Figure 4. On the left side of the figure, the inode has multiple hard links (indicated by the value of `i_nlink`) in the file system `FS1`. If one wants to clone `FS1` to `FS2`, Prism must replicate the inode to `FS2` and rebuild the hard link structure. On the right side, the inode only has one hard link in `FS1`. When `FS1` is cloned to `FS2`, Prism can directly create a new hard link to this inode in `FS2` (indicated by $i_nclone = 2$).

When implementing the Prism system, we always try to reuse the existing Linux and ext3 file system code to expedite the development and avoid incompatibility issues. Like most file systems that support hard links use *reference counting*, the ext3 file system maintains an integer value as the `i_nlink` field in each file's inode that describes the file attributes and data blocks associated with the file. This integer represents the total number of links that have been created to point to the data. When a new link is created, this value is increased by one. When a link is removed, the value is decreased by one. The maintenance of this value assists users in preventing data loss. It is also the simplest way for the file system to track the use of a given area of storage, as zero values indicate free space and nonzero values indicate used space.

Unfortunately, the `i_nlink` field alone is insufficient for Prism to tell whether an inode has multiple links within the parent filesystem. An inode may have no more than one hard link in a local file system even if the `i_nlink` field is greater than one. For example, in the right side of Figure 4, after `FS2` clones `FS1`, the inode's `i_nlink` field is equal to two. But this inode has no hard link in any local file system. If `FS1` is cloned to `FS3`, the inode can still be directly hard linked without replication.

In order to tell whether an inode has more than one internal hard links, we add a field `i_nclone` to the inode structure. This field is initialized to one and indicates the number of clones that share the file. The above restriction ensures that Prism always holds the following invariant:

> An inode's `i_nclone` value is always equal to `i_nlink` value unless the `i_nclone` value is equal to 1.

With the `i_nclone` field, it is simple to determine whether an inode has more than one internal hard links: an inode have internal hard links if and only if the inode's `i_nclone` field is equal to one and the `i_nlink` field is greater than one.

To clone a filesystem `FS1` to `FS2`, Prism traverses the parent filesystem's directory and recursively clones each encountered filesystem object:

- For a directory, create a directory with the same name at the corresponding place in the cloned file system.
- For a file whose inode only has *one* internal hard link in `FS1`, uses the standard ext3's hard link function to create a new hard link to the inode, and increases the inode's `i_nclone` by one. This step only involves an update to the original inode and no data copy is required.

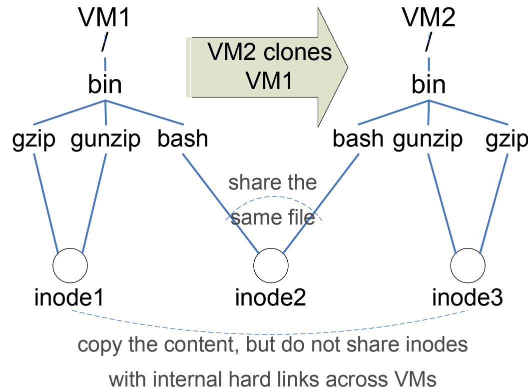

Figure 5: Example of a parent and child fileystems after cloning

- For a file whose inode has *more than one* internal hard links, Prism replicates this inode to `FS2`, and records both the original inode number and the new inode number. In the following cloning procedure, if the other hard links of the original inode are encountered, Prism creates a standard ext3's hard link to the new inode so that the hard link structure can be rebuilt. Note that Prism does not increase the `i_nclone` field.

Figure 5 shows an example of a directory tree that is cloned, where one of the files has multiple internal hard links to it, prior to cloning and after cloning.

Prism processes reads to a cloned filesystem as standard ext3 filesystem. However it takes different steps to deal with writes to a pext3 file:

1. If the inode's `i_nclone` value is equal to one, this inode is not shared by shared by other filesystems. The file is the same as in a normal ext3 file system. Prism can safely change this file in place, as standard ext3 filesystem does. Neither standard hard link semantics nor cloning semantics will be broken.

2. If the inode's `i_nclone` value is greater than one, this inode is cloned and shared by more than one local filesystems. Prism replicates the inode to a new one. Next, Prism updates the file entry to point to the new inode in the local filesystem from which the file is to be modified. Beyond this point, the original inode is not shared by the parent filesystem and its clone. All subsequent changes are made to the new file and are transparent to other local filesystems, which keeping the cloning semantics. It is interesting to note that the `i_nclone` must be equal to `i_nlink` in this case. The file cannot be pointed by more than one internal hard links in any sharing filesystems. Therefore, no hard link semantics will be broken.

It is interesting to describe how Prism deals with the `ln` command on a file shared by multiple filesystems. This command attempts to create a new internal hard link to the file. Because the file is shared by multiple filesystems, Prism replicates it and point the new hard link to the new copy. Moreover, to preserve the standard hard link semantics, the original file entry must be changed to point to the new inode as well. In practice, Prism does not need to scan the entire filesystem to locate the original file entry, because the `ln` command specifies the source path which is just the original file entry.

4.2 Dynamic Cloning

While the clone-by-sharing technique significantly reduces the overhead of cloning a filesystem, the file level cloning mechanism is still slower than block-level cloning. The main reason is that the file-level cloning

needs to traverse the filesystems and examine each encountered object. In contrast, most block-level cloning only build a block mapping table without interpretation. While scanning each filesystem object is necessary to provide users more flexibility in cloning, it does incur substantial overhead. 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 19K directories. The total size is around 6G bytes. Prism spent approximately 54 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 of the 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 a dynamic cloning mechanism for Prism. For easy comparison, we call the cloning mechanism described in previous section ”static cloning”.

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

While the dynamic cloning scheme is conceptually simple, it is substantially complicated to enable users to access files during cloning. In particular, the dynamic cloning mechanism must properly address the following two situations:

1. A user may access a file in the cloned filesystem that has not been cloned. As the dynamic cloning mechanism has presented the user an illusion that the cloning has been done, Prism should be able to quickly respond to the file request, rather than blocking the user until the file is eventually encountered and cloned by the background thread.

2. A user may attempt to modify a file in the parent filesystem that has not been cloned. Suppose the modification is made before the file is cloned. When the background thread clones this file, it will actually clone a file whose content is different from the snapshot when clone command returned. However, from a user perspective, it is reasonable to think that the cloning is complete when the clone command returns. Users would expect that the content of the cloned file is same as the one presented when the cloning task is finished. The parent and clone system are supposed to behave like isolated systems from that point. Therefore, the dynamic cloning mechanism must ensure that a file in the parent filesystem is cloned before being modified.

4.2.1 On-Demand Cloning

To address the first problem, the cloning mechanism must dynamically adjust the cloning order on demand. We develop a 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 with the steps described in Section 4.1. However, it clones a subdirectory by creating an empty subdirectory at the corresponding location in the clone. In other words, this function only clones one level of directory hierarchy, and will not go deeper into subdirectories. A subdirectory cloned in this way is flagged as “UNEXPANDED” and associated with the corresponding source directory’s inode number. We added 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, these two fields indicate whether the directory is expanded or not. After a directory is expanded, it is flagged as “EXPANDED”. All subdirectories are empty and marked as “unexpanded”.

We extend the ext3 permission function to implement the dynamic VM cloning. When a process attempts to access a directory, pext3 file system always calls the permission function to check that the process has

sufficient right to access the directory. In the permission function, Prism determines whether the directory is expanded or not by checking the directory's `expanding_flags` field. If the `expanding_flags` field is equal to $TRUE$, the directory is already expanded. The permission function quickly jumps to the normal permission checking routine. Otherwise, the directory is not expanded yet. Prims calls the `pext3_expand_dir` function to expand the directory.

Based on this design, Prism uses a kernel thread to clone a filesystem at background. When a user request to clone a filesystem, Prism simply expands the parent filesystem's root directory and mark the root directory as "UNEXPANDED" in the cloned filesystem. Next, Prism starts the background cloning thread and then returns, presenting the user an illusion that the cloning task is complete. Therefore, the cloning latency sensible to users is only the time used to clone the entries under the parent filesystem's root directory. Usually, it is less than one second. The background thread is essentially a directory walker that starts from the root directory to traverse the entire filesystem. For each directory, the directory walker emulates to lookup a file under that directory. The only purpose of the lookup operation is to trigger the permission checking function to expand the directory if it is not expanded. If a user attempts to access a file that is not cloned, the pext3 filesystem must walk along the specified pathname to locate this file first. After the cloning thread recursively traverse the cloned filesystem, the permission checking function will be triggered to expand all directories, which accomplishes the dynamic cloning task.

Another merit of this design is that on-demand cloning can be easily supported. Suppose a user attempts to access a file `/a/b/c`, but only the root directory `/` has been expanded. Prism will behave as a normal filesystem to walk through the path

$$\underline{/} \rightarrow \underline{/a} \rightarrow \underline{/a/b} \rightarrow \underline{/a/b/c}$$

to get the file $\underline{/a/b/c}$'s inode for accessing. This is essentially a walking thread parallel to the background cloning thread. Prism can seamlessly adjust the cloning order to aggressively clone the directories needed for the file request. This easily realizes the on-demand cloning feature.

4.2.2 Preserving Cloning Semantics

If a user issues a clone command and instantly gets a shell prompt, it is reasonable for this user to believe that the filesystem cloning is finished. According to the cloning semantics, any modification made to a filesystem (parent or clone) beyond that point should be transparent to the other. For static cloning, it is easy to meet this requirement. Because the parent and clone are suspended before the entire cloning job is done. However, with the dynamic cloning mechanism, a user will get a usable filesystem before the cloning is finished. A user can change a file in the parent file system before the file is cloned. This can break the cloning semantics if no synchronization mechanism is applied.

Prism employs an aggressive cloning mechanism to deal with this situation. If a process attempts to write a file in the parent file system that has not been cloned, Prism blocks this modification request, aggressively clones the file, and then processes the modification request. An important step in the above procedure is to tell whether the file is cloned or not. There are two major challenges in identifying a file's clone status: *efficiency* and *limited space*. Prism must efficiently tell whether a file is cloned or not, because this is critical to the filesystem performance. A naive way to keep tracking of the cloning status is to keep a list of the files that have been cloned. By looking up the list, Prism can determine a file's clone status before modifying the file. However, this solution can be slow when the file list is large. It would be more efficient if Prism can determine a file's cloning status with the information associated with the file. However, there comes the second challenge: pext3 stores a file's metatdata in its inode. But there is only small spare room in an inode.

To address this problem, Prism maintains a global logical timestamp to record the occurrence time of cloning events. The logical timestamp is an 32-bit unsigned integer and is initialized to zero. This logical timestamp is incremented at the beginning of each clone task. In addition, Prism adds a 4-byte field, `i_lastclone`, to the `pext3` inode structure. This field stores the logical timestamp, indicating when is the last time the inode is cloned. Note that an inode pointed by multiple internal links is not regarded as "cloned" until the entire hard link strucuture are rebuilt in the clone. In addition, each local file system is associated with a `clone_start` timestamp. The `clone_start` timestamp is normally equal to zero. When

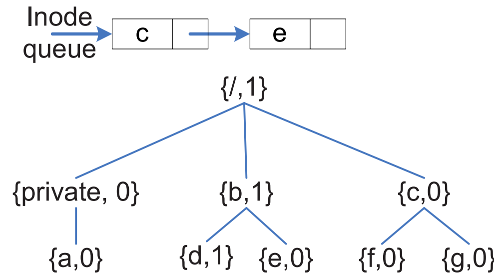

Figure 6: Clone procedure

Prism starts to clone a filesystem $FS1$ to $FS2$, it sets the `clone_start` timestamp is set to current global logical timestamp value. When the entire cloning task is finished, Prism resets the `clone_start` timestamp back to zero.

Prism is able to determine whether an inode is cloned or not by comparing the parent filesystem's `clone_start` timestamp with the inode's `i_lastclone` field. Suppose Prism is cloning a filesystem $FS1$ to $FS2$, and a process attempts to modify a file in the parent filesystem $FS1$. If the file's inode has an `i_lastclone` field that is less than $FS1$'s `clone_start` value, the inode is not cloned yet. The file modification request should be blocked until the inode is cloned to the child filesystem $FS2$. It is important to note that Prism only allows one dynamic clone task at a time. Supporting multiple cloning tasks on a single filesystem has a lot of synchronous issues and is not currently supported.

The above design can be complicated by one exception. Suppose an inode is pointed by three internal hard links (for example, `/a`, `/b`, and `/c`)in a filesystem $FS1$, which is being cloned to a new filesystem $FS2$. Suppose Prism has cloned `/a`, but has not rebuild the hard link structure for `/b` and `/c`. Now a process attempts to remove file `/b`. If the inode's `i_lastclone` field is updated to $FS1$'s `clone_start` value when `/a` is cloned, then Prism will mistakenly believe that the inode has been fully cloned and delete `/b`. Unfortunately, this is wrong because the file `/b` has not been cloned to $FS2$ properly. Therefore, Prism must delay updating an inode's `i_lastclone` field until all its hard links are rebuilt in the cloned filesystem. In the above case, Prism does not need to block the writing process until all three hard links are rebuilt. In fact, it could take long time for the background thread to process the file `/c` if this file locates far away from the first two hard links. Prism can process the file deletion as long as it will not change the cloning semantics. In this case, once Prism aggressively rebuilds the hard link for the file `/b`, it can safely delete this file in $FS1$.

It is possible that the cloning thread will never be able to fully rebuild the hard link structure of an inode. In the above example, if the hard link `/c` is excluded from cloning by the administrator, Prism cannot rebuild the entire hard link structure. Fortunately, this will not affect the normal running of the cloning mechanism. Once the filesystem is cloned, Prism sets the filesystem's `i_lastclone` back to zero and cleans up the in-memory structures used to rebuild the hard link structure. Subsequent changes to the inode in the parent filesystem will not be affected then.

In summary, the above mechanism helps Prism to efficiently determine whether an inode has been properly cloned or not. Then, this mechanism can aggressively resolve the conflict between filesystem cloning and accessing. This enables a process to safely access a file in either parent or cloned filesystem even if the entire cloning task is not complete.

An example shown in Figure 6 helps illustrated the dynamics in filesystem cloning. In this example, the `/private` directory is excluded from cloning. All timestamps are initially set to be 0 in this example. Upon a clone request, the Prism server increments the global timestamp to one and sets the `clone_start` value for the parent and child filesystems to 1. Then, a background thread starts to perform cloning. Control is returned to the user at this point. The background thread maintains a queue of inodes to clone next. In the

situation shown in Figure 6, the following inodes have been processed: /, /b, and /b/d. The cloning thread has updated their i_lastclone values to 1. When the dynamic process completes, all inodes, except those under /private, will have a i_lastclone value of 1. The clone_start of the filesystem will be set back to 0.

From the performance perspective, Prism avoids blocks a writing process for a long time by aggressively resolving conflicts between cloning and accessing. For example, in Figure 6, a process attempts to write the file g. Prism blocks this request and actively clones the filesystem objects along the path from the root the file g. In this example, Prism actively clones directory c and its children files f and g. As such, the clone of file g is effectively prioritized. After that, Prism can safely unblock the writing process and write file g.

n interesting situation is where a process attempts to update a file that is excluded from cloning, for example, /private/a in Figure 6. This request will be blocked because this file's i_lastclone value is 0 and is less than the filesystem's clone_start value. The file /private/a is then added to the prioritized queue. But the cloning function is able to find that the file /private/a is not subject to cloning. It then updates the file's i_lastclone value to the filesystem's clone_start) value, indicating that the file is free to modify. Note that any other files under /private need not be added to the queue and will continue to have a cloneTimstamp of 0.

The one restriction we place during cloning is that we do not cross mount-points. Note that each clone is considered as a filesystem in Prism and can be mounted by clients. Suppose that a user has a filesystem $FS1$ on which a filesystem $FS2$ is mounted at a directory in $FS1$. If $FS1$ is cloned, we will not recursively clone $FS2$, even if it is managed by Prism. This restriction is currently necessary to make dynamic cloning fast. The Prism clonefs command uses its first argument to determine the parent filesystem and updates the filesystem's clone_start to ensure a consistent snapshot. To support recursive cloning, we will need to identify all filesystems that are mounted within the parent filesystem and ensure that they also undergo dynamic cloning, prior to returning control back to the user. Our current implementation does not do this, though it could be a future extension of the system.

Overall, dynamic cloning has the advantage that both the parent filesystem and the cloned filesystem are usable immediately without requiring a traversal of the directory structure. However, the access performance can be lower than normal if one attempts to access a file that is not cloned. We found that the delay was not really significant for all practical purposes because of the queue prioritization, and much better than the alternatives of copying the filesystem (which could take minutes to hours for a large filesystem) or freezing the filesystem and doing static cloning (which could take seconds to minutes for a large filesystem). After the cloning task is done by the background thread, filesystem access speed will resume to normal.

4.3 Lazy Cloning

Prism also provides another cloning mode called *lazy cloning*. In lazy cloning, the background cloning thread only processes inodes on-demand. In other words, an inode is cloned only if it is accessed by a process. The major advantage of this mode is that it only consume little system disk and CPU resources. The system does not need to pay overhead to clone files that are never accessed. This is particularly useful for applications that only need a ephemeral filesystems. Software testing is a good example. Users often tend to destroy the clone after they test an application. It is often unnecessary to clone the entire filesystem for such an ephemeral system. Moreover, users can choose to preserve cloning semantics or not. If a user choose not to keep the cloning semantics, the cloning overhead can be further reduced, as Prism does not need to aggressively clone a file when it is only accessed in the parent filesystem.

The lazy cloning mode is also useful for evaluating the access performance of a dynamically cloned filesystem. It gives a worst-case performance bound because the lazy cloning maximizes the waiting time of file requests.

4.4 Additional Optimization: Block-level Copy-on-write

As described in Section 4.1, Prism allows a clone to share the unchanged files with its parent. However, if either one attempts to modify a shared file, copy-on-write must be performed to create a new copy so

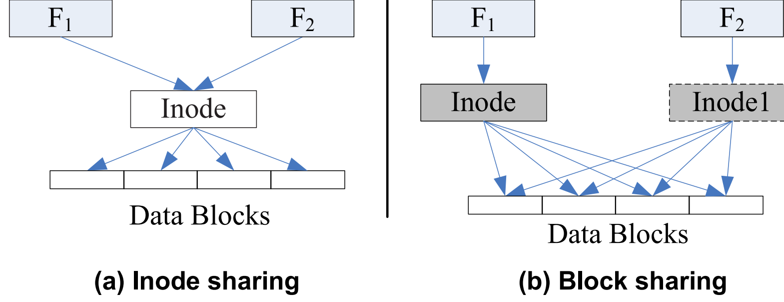

Figure 7: Prism's Two-level Sharing Mechanism

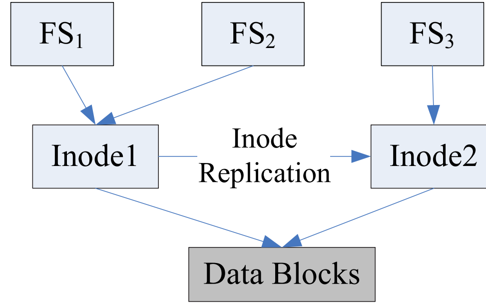

Figure 8: Two-level File Sharing

that subsequent modifications can be made to the new copy. As such, Prism preserves the isolation property between VMs. Conceptually, we can think copy-on-write and replication in the above cloning procedure as the same as copying a file entirely, including the inode and associated data blocks. However, it is impractical to implement copy-on-write and inode replication this way, as this could be very inefficient for some file operations. For example, suppose one issues the `chmod` command to change a file attributes. Because the attributes are only stored in the file's inode, Prism does not change the data blocks at all. However, with a naive copy-on-write mechanism, Prism would have to replicate the file's inode as well as data blocks, which can be slow for large files. Therefore, an efficient copy-on-write mechanism is needed.

Ideally, the copy-on-write mechanism should only copy the modified part to the new copy and still share the unmodified part. This would require to break a file into multiple parts and allow each part to be individually shared. To avoid loss of data, appropriate mechanism must be devised to track the reference count of each part.

In Prism, a file consists of two parts: *inode* and *data blocks*. The inode describes the file's attributes and data block numbers, while the data blocks store the file content. As shown in Figure 7, Prism allows two filesystems to share data at two levels: the inode level and the data block level. The inode level sharing is also abbreviated as *inode sharing*. As shown in Figure 7(a), if two filesystem objects share a same inode, they essentially share everything described by the inode, including file attributes and content. The block level sharing, however, is different. As shown in Figure 7(b), two inodes can contain different attributes. But their associated data blocks can be partially or fully shared/overlapped. This sharing mode ensembles block level sharing and is also referred to as *block sharing*.

An example shown in Figure 8 shows how the two-level sharing mechanism helps improve Prism's copy-on-write performance. Suppose FS_1, FS_2, and FS_3 initially share $inode1$ with inode sharing mode. When FS_3 requests change the file permission, Prism replicates $inode1$ to $inode2$, but making $inode2$ share the same data blocks with $inode1$. Then, Prism changes file attributes on $inode2$. As such, FS_3 owns its own file attributes while still sharing the identical file content with other filesystems. In this example, FS_1 and FS_2 shares $inode1$, which is inode sharing. $inode1$ and $inode2$ share the same data blocks, which is block

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
Postmark	version 1.5
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

Table 1: Experimental platform

sharing.

4.5 Tracking Reference Counts of Shared Objects

Based on the two-level sharing model, a file's inode and data blocks can be shared separately. It is crucial to track the reference counts of each sharable component to avoid loss of data.

Prism uses an inode's `i_nlink` field to track the inode's reference count. If an inode's `i_nlink` value is equal to zero, this inode can be freed and its associated data blocks will not be referred to this inode. Note that the data blocks can still be shared by other inodes.

To track the reference count of data blocks, Prism deploys a block reference count table. Each table entry is one byte long, and corresponds to a 4KB data block. A data block's reference count records the number of *inodes* that shares this block. For a 30GB disk partition, its reference count table will be around 15MB, occupying 0.05% of disk space. Each block can be shared by at most 255 inodes, which is sufficient in most scenarios. If an inode's `i_nlink` value is reduced to zero, Prism will remove this inode and reduce the reference counts of all data block associated with the inode by one. If a data block's reference count is reduced to zero, Prism will free this data block and reclaim the disk space.

It is possible that more than 255 inodes attempt to share a same data block. Because each block reference count is only one byte long, it cannot handle this situation. Prism currently addresses this problem by performing copy-on-write operations if a block's reference count will exceed 255. While this solution incurs additional data copying overhead, it will not substantially impact the overall performance because reference count overflow is rare in a real world system.

5 PERFORMANCE EVALUATION

Table 1 describes our development 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 distribution of Linux. We compared the performance of `pext3` (Prism-enabled ext3 kernel module) with the native `ext3` installation. Both systems were accessed locally from test applications. 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.

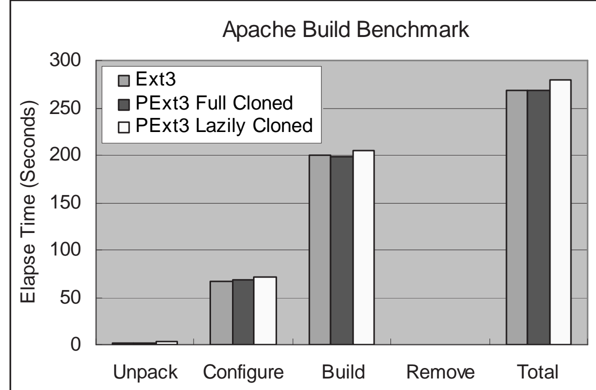

Figure 9: Performance of Apache build workload. "Ext3" stands for standard Ext3 file system;"PExt3" stands for Prism modified Ext3 file system

5.1 Static and Dynamic Cloning Latency

We first evaluated the performance of static and dynamic cloning. For the parent filesystem, we used a filesystem consisting of the entire Fedora Core 4 system with standard software packages, including around 170K files and over 17K directories.

We first compared the cost of full copying versus static cloning. The full copy of the file system took over 10 minutes (630 seconds), while static cloning took 54 seconds, including rebuilding the hard link structure and establish block sharing.

With the dynamic VM cloning mechanism, the cloning activity largely occurred in the background; Prism instantly presented the users with an accessible file system. The observed latency was 0.18 seconds.

5.2 Performance on the Apache Workload

To evaluate the penalty of dynamic cloning, we used an Apache build task as a representative of typical workloads on a normal development machine. To get the worst-case performance for Prism's dynamic cloning mechanism, we used the "lazy" cloning mode — Prism only clones file on-demand and does not run background thread to clone the unvisited files. As another comparison point, to get the best-case performance for Prism, we allowed the background cloning to complete (which takes approximately 54 seconds). We compared ext3, Prism lazily-cloned, and Prism fully-cloned on the workload.

We used Apache 2.0.58 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.

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 file system. For the `pext3` experiments, We guaranteed that by using "`chroot`" [12] to the cloned file system before running the benchmark. As a result, all input and output files needed for the benchmark are accessed from the cloned file system. To avoid warm cache effects caused by previous runs, we always ran the experiments right after the file system was mounted.

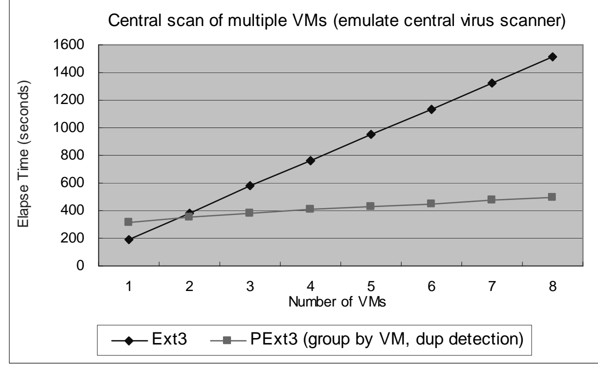

Figure 10: Performance of central scanning 8 VMs. "Ext3" stands for fully copied file systems. "PExt3 (group by VM, dup detection)" stands for cloned file system.

In Figure 9, 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 file system. With a fully cloned filesystem, the performance difference with `ext3` was negligible.

5.3 Postmark Benchmark

We also ran the Postmark (version 1.5) benchmark on a lazily cloned Fedora Core 4 file system. The Postmark benchmark first created 10000 files whose size varied from 512 bytes to 40K bytes. Next, the benchmark performed 20000 transactions consisting of file reads, writes, creates, deletes, and finally it removed all files. The experimental results showed that the Postmark benchmark took 54 seconds on a fully copied `ext3` file system on the guest and 55 seconds on a lazily cloned file system. The lazy cloning penalty is surprisingly low, because the Postmark benchmark mainly operates on newly created files in one parent directory. Once the parent directory is replicated (some delay due to lazy cloning), the Prism filesystem behaved largely like an ext3 filesystem. Any files created within the parent directory are local to the clone, and thus do not incur any additional delay.

5.4 Central Scan of Multiple Clones

Prism is designed to improve the manageability of multiple cloned filesystems. One of the applications that we suggested in Section 2 was central scanning of multiple clones, for virus checking as one example. To see if Prism could provide performance advantages, we cloned the parent system up to 8 times.

We then scanned n cloned systems and compared the performance with scanning n copied filesystems, both sequentially (in `ext3`). Both the fully copied file system and the cloned file system have 170K files and 17K directories. The Prism central scanning tool maintained a list of scanned inodes (retrieved via the Linux `stat()` call) in a hashtable. If it encountered the same inode again from another filesystem and the inode has the same modification time as the recorded value, it did not rescan the file content because the inode was not modified. This guaranteed that any file that was not modified in any of the clones was scanned only once.

Figure 10 shows the central scanning performance on both pext3 and ext3 filesystems. For $n = 1$, `ext3` outperformed `pext3` as every file had to be scanned in both systems. 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. 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 significantly lower.

5.5 Smartdiff: comparing two cloned filesystems

If a user runs an untrusted application on a cloned filesystem, he or she might want to know what files are modified by this application. In a standard Ext3 file system, the user can run command "`diff -r`" to compare the cloned file system with the parent file system. With Prism, we are able to write a `smartdiff` program to do the same task more efficiently. The `smartdiff` program compares two filesystems like the standard `diff` tool. But it detect files that are not changed since being cloned by comparing their inode numbers. These files can be skipped without further checking.

To compare the performance of these two mechanisms, we set up a parent file system F_{base} containing the standard Fedora Core 4 distribution. Then, we fully copied this file system to create a new file system F_{copy}. Next, we cloned F_{base} to create a cloned file system F_{clone}. On each of the three file systems, we unpacked the Apache 2.0.58 package to emulate an untrusted program.

We then ran the following commands to compare the performance.
`diff -r` F_{base} F_{copy}
and
`smartdiff -r` F_{base} F_{clone}

The `diff` program took 31 minutes, while `smartdiff` took only 3 minutes, providing over 10 times improvement.

6 RELATED WORK

The capability of data cloning and isolation have been developed in some existing *block-level storage systems* and *file-level storage systems*. We first discuss the block-level storage systems. Then, we examine the file-level storage systems. Finally, we discuss the possibility of combining a block-level scheme with a file-level scheme to provide these features.

6.1 Block-level Storage Systems

Most virtual machine systems today, e.g., Xen [1, 28] and VMWare [26], support an abstraction of a virtual disk. The virtual machine monitor (VMM) layer manages the physical storage device, partitioning it into blocks. It maps a virtual disk to a set of underlying blocks on the physical device. The virtual disk appears like a regular physical disk to a guest virtual machine.

Full cloning of the virtual disk for the purpose of snapshots is relatively easy to provide in this architecture. The VMM layer can clone a new virtual disk from an existing virtual disk by mapping the clone's blocks and the existing disk's blocks to the same physical blocks. The only cost is updating the reference counts of all the physical blocks and initializing a new map (the cost is not insignificant, but reasonable). To guarantee isolation, copy-on-write at the physical block-level is used whenever a block has multiple references to it. VMware [26, 23] and Parallax [27] use this model to support quick snapshots. Outside of the virtual machine world, some business storage products like NetApp's Data ONTAP™ [9] also support cloning disks or storage volumes by sharing data at block level. These systems share the same advantages and disadvantages with virtual disks.

One advantage of virtual disk cloning over the Prism model is that the virtual disk cloning can be file-system agnostic. It is possible to clone an arbitrary virtual disk, irrespective of the type of file system that resides on it. On the other hand, Prism's cloning model has advantages in terms of rapidly cloning any part of a file system. Virtual disks must be cloned in their entirety. It is difficult to exclude certain files that the user considers private from being cloned, because the block-level storage systems have no file system knowledge.

In contrast, Prism uses copy-on-write in the filesystem at the file-level. When files or inodes are modified, block-level copy-on-write is also used. This allows central data management such as virus scanning on multi-

ple overlapping clones to be implemented efficiently. As far as we know, current virtual disk implementations do not provide such a capability.

6.2 Network File Systems

Network file systems, such as NFS [15], AFS [8], and Samba [24], can be shared among multiple operating systems without cloning. Collective[3, 19, 20], for example, suggested to use this architecture to provide a shared, read-only software environment for client machines interconnected over the network. But these systems do not provide the same semantics as a cloned filesystem. If a filesystem is shared in read-only mode, it guarantees isolation, but does not provide users with the ability to write to the system. If it is shared in read-write mode, it does not provide isolation guarantees. Prism provides the performance of sharing a common filesystem, but the semantics of an isolated filesystem for each user.

6.3 File Systems for Virtual Machines

The Ventana[17] system extends distributed file systems to improve VM manageability and flexibility. Ventana manages a VM's file system as a collection of directory branches, with the branch type being either public or private. A public branch can be modified by any sharing VM. The modifications immediately become visible to all sharing VMs. A private branch can be read by multiple VMs, but can only be modified by the owner VM.

Prism differs from Ventana in that it provides a very efficient cloning operation. The Ventana paper mentioned that a file system branch is created by *copying* an existing branch, but it does not discuss the details on how to efficiently *copy* a branch.

The data management architectures of Prism and Ventana are different. Prism manages data under the standard file system framework, Therefore, it can take advantage of heavily optimized file system facilities such as the metadata cache and the page cache to improve system performance. Ventana uses object-based storage. Performance optimization may require additional efforts, which are not discussed in the Ventana paper.

The VMFS [25] file system, shipped with the VMware ESX Server, is designed as a high-performance cluster file system that provides storage virtualization. ZFS [13] for Solaris 10 is designed to provide a similar capability to that of VMFS. They both provide fast cloning support at the block level. But, as far as we can tell, neither claims to provide selective cloning or is able to efficiently compare clones with each other.

6.4 Versioning File Systems

Some file systems provide a data versioning feature. Examples include Cedar [6], Elephant [18], CVFS [21, 22], 3DFS [10], VersionFS [14], Clotho [5], WAFS [7], and Ext3COW [16]. The versioning techniques can be potentially extended to provide efficient cloning. For example, Ext3COW allows a file's multiple versions to share identical data blocks.

Prism differs from versioning file systems in several aspects. First, Prism is able to provide users multiple mutable filesystems. After the clone command returns, both the parent filesystem and the clone can be accessed as two independent filesystems. Each clone can be changed without affecting other clones. This requires the support of synchronization and isolation between two filesystems. In contrast, versioning file systems only allow users to modify current version of files. Past versions of files are read-only. Second, Prism provides a simpler interface to the user. Users see the same familiar namespace of a standard filesystem and cloning is conceptually similar to copying a filesystem, except much faster. Third, Prism is not designed as a versioning file system. Although each clone can be treated as a version, Prism does not provide a version retaining mechanism to determine when a version can be removed to reclaim disk space. Finally, there are obvious differences in terms of inode structure (Prism's changes are modest), naming (Prism is simpler since clones appear to users like directories), and details like reference count management.

Alcatraz [11] is a system that allows multiple users to share a file system in read-only mode. If a user needs to change a shared file, Alcatraz produces a new copy of this file with the copy-on-write technique.

The new copy is stored in this user's private space. Alcatraz maintains a mapping table to help the user to locate the private copy. By manipulating the mapping in file system name space, Alcatraz can potentially achieve data sharing and isolation. Alcatraz maintains the name-to-file mapping in memory, which is only good for a small and temporary system. Prism is more flexible as it allows cloned filesystems to be also cloned themselves. They are managed just like the parent filesystem. It is also more efficient in terms of using copy-on-write at multiple levels and supporting hard links.

7 CONCLUSION

This paper presented the design of Prism, a filesystem that allows selective cloning of any part of the directory structure and provides isolation between the original copy and the cloned copy. Prism utilizes a dynamic cloning scheme and is implemented as a kernel module on Linux by modifying `ext3`. When cloning filesystems, even containing hundreds of thousands of files such as the Fedora distribution, both the parent and the child filesystems are usable within a few hundred milliseconds.

We showed that a selective cloning operation can be highly desirable in filesystems for many important tasks, such as preventing untrusted code from damaging the filesystem and creating new filesystems for virtual machines from existing ones, while protecting home directories and other private content. Furthermore, when cloned systems have a high degree of similarity, Prism helps make centralized scanning or comparison operations for multiple cloned file systems significantly faster than scanning or comparing each file system individually.

On the Postmark benchmark and the Apache build workload, Prism's performance is comparable to that of a standard ext3 filesystem. For applications that require scanning or comparing multiple cloned filesystems, Prism was found to be significantly faster. On a workload of comparing two filesystems, Prism provided over 10 times of performance improvement.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[2] D. P. Bovet and M. Cassetti. *Understanding the Linux Kernel (Ed. A. Oram)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[3] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.

[4] D. Euresti. Self-certifying filesystem implementation for windows. Master's thesis, Massachusetts Institute of Technology, August 2002.

[5] M. D. Flouris and A. Bilas. Clotho: Transparent data versioning at the block i/o level. In *Proceedings of NASA/IEEE Conference on Mass Storage Systems and Technologies*, pages 315–328, Ottawa, Ontario,Canada, 2004.

[6] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The cedar file system. *Commun. ACM*, 31(3):288–298, 1988.

[7] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an nfs file server appliance. In *USENIX Winter*, pages 235–246, 1994.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[9] M. Klivansky. A thorough introduction to flexclone^{TM}volumes. Technical Report TR3347, Network Appliance Inc., October 2004.

[10] D. G. Korn and E. Krell. A new dimension for the unix file system. *Softw. Pract. Exper.*, 20(S1):19–34, 1990.

[11] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *ACSAC*, pages 182–191, 2003.

[12] R. McGrath and Free Software Foundation. Chroot c run command or interactive shell with special root directory. The Linux Manual Pages.

[13] S. Microsystems. Solaris zfsthe most advanced file system on the planet. 2007. *http://www.sun.com/software/solaris/ds/zfs.jsp*.

[14] K. Muniswamy-Reddy, C. P. Wright, 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)*, pages 115–128, San Francisco, CA, March/April 2004.

[15] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceeding of USENIX Summer 1994 Technical Conference*, pages 137–152, 1994.

[16] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transcations on Storage*, 1(2):190–212, 2005.

[17] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *3rd Symposium of Networked Systems Design and Implementation (NSDI)*, May 2006.

[18] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[19] C. P. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *LISA*, pages 181–194, 2003.

[20] C. P. Sapuntzakis and M. S. Lam. Virtual appliances in the Collective: A road to hassle-free computing. In *HotOS*, pages 55–60, 2003.

[21] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in versioning file systems, 2003.

[22] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proc. of Operating Systems Design and Implementation (OSDI)*, pages 165–180, 2000.

[23] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[24] S. Team. Samba: Opening windows to a wider world. Sept 2006. *http://us1.samba.org/samba/*.

[25] VMware. Vmware vmfs: High-performance cluster file system for storage virtualization, Oct 2006. http://www.vmware.com/pdf/vmfs_datasheet.pdf.

[26] VMware Inc. Vmware success stories. Sept 2006. *http://www.vmware.com/customers/stories/*.

[27] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. 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.

[28] XenSource Inc. Amazon launches xen-powered virtual datacenter on demand. Nov 2006. *http://www.virtualization.info/2006/08/amazon-launches-xen-powered-virtual.html*.