Improving Virtual Appliances through Virtual Layered File Systems

Shaya Potter Jason Nieh
Computer Science Department
Columbia University
{spotter, nieh}@cs.columbia.edu

Abstract

The problem of managing computers is growing in complexity due to the increasing amount of physical and virtual computers that one has to administer as well as the varying roles that those computers fill. As each machine is effectively fully independent, the amount of work an administrator does scales linearly with the amount of machines. In order to solve this problem, we introduce Strata, a system that enables new ways of managing these many distinct installations. Strata enables many systems to be easily managed by introducing a virtual layered file system that composes individual software layers together into a single file system view. By providing the ability to share layers between installations, Strata eases the creation of independent systems and opens up new ways to use computers. We have implemented Strata on Linux without requiring any application or operating system kernel changes. Our measurements on real world applications demonstrate that Strata imposes little overhead.

1 Introduction

As computing systems become more complicated, the burden they place on administrators tasked with their care increases. Application service providers are beginning to become common and the market for them is expected to grow quickly. As they expand, their management load will increase as they provide applications and services to many clients around the globe.

One way administrators have tried to lower the administrative load is by leveraging virtual appliances. Virtual appliances attempt to solve two main problems. First, they attempt to make provisioning of systems a simple task. Second, they attempt to simplify the updating of provisioned systems. Some virtual appliances, due to their architecture, also solve an ancillary problem of enabling easy upgrade rollbacks if it is determined that an upgrade breaks or degrades a user's system.

Virtual appliances solve these problems by leveraging virtual machines and the virtual disks they provide to provide template virtual machines that can be copied and used by users. In order to enable a virtual appliance to be upgraded, the virtual appliance creator will generally divide the disk into two distinct areas. The first area, the shared base *System*, can be viewed as the actual virtual appliance and is completely controlled by the virtual appliance creator. Every user that makes use of this virtual appliance shares this disk in what is effectively a read-only manner. The second part, the personalizable *Data* area, on the other hand, is completely controlled by the user. This area is where all the user's modifications to the file system will be stored. In a traditional Unix system, one can view this as the majority of the file system is System, while the home directory tree is Data. By dividing the file system in this manner, the virtual appliance creator can update the System area while preserving the contents of each user's virtual appliance's Data area. By splitting the file system, one can even downgrade to the system's state as it existed at a previous point in time as its fully independent from the data of the user using this particular instantiation of the virtual appliance.

However, virtual appliances suffer from a number of problems. First, since they run within virtual machines, they need a physical machine to host them and this physical machine needs to be maintained as well. Since virtual appliances are based around virtual disk drives that can be shared, this model can not easily be extended to physical machines. Second, because virtual appliances are based on a shared block device or file system, the upgrade process does not easily scale when one has to support many distinct installations. Since each distinct virtual appliance will have a distinct virtual system disk, each distinct virtual appliance disk must be upgraded independently. While ideally an administrator would only support a single virtual appliance, since users need access to different sets of applications, the administrator will end up having to support a virtual appliance for each set of applications one's users need.

The fundamental problem with virtual appliances is that they are based on the concept of a monolithic file system or

block device. What this means is that they view the storage entity, be it a file system or a block device, as a set of data at specific point in time. It is possible to create changes to this entity, for instance, when one upgrades it or modifies its files, without losing the previous state. However, the file system is still a monolithic entity, just that one can access the file system at different points in its time line, since the previous state was not lost.

Strata therefore attempts to solve three main problems to improve the ability of administrate large numbers of virtual and physical systems. First, Strata enables systems to be easily provisioned. Today, an administrator might rely on only supporting a single image that can be provisioned on any system by various methods. However, this restricts the ability of an administrator to differentiate systems.

Second, Strata enables large numbers of systems to be easily upgraded. Upgrading large numbers of systems can become an orthogonal problem to provisioning systems. If the system can not be modified, upgrading can simply be a matter of modifying the master image. On the other hand, most systems are modified in some ways, be they simple configuration issues or additional software that is installed. This differentiation can prevent easily upgrading all instances that an administrator manages.

Finally, Strata enables piece meal rollback of a software on a system. Strata requires this because just because an administrator wants to upgrade a piece of software, due to a new version being available or a security hole that has to be fixed, it does not mean he wants to lose access to the old version. For example, the administrator might want to run the old version in parallel with the new version for testing. The administrator might want to keep the old version readily available if the new version fails. Similarly, since new versions of software are not always 100% backwards compatible, an administrator might want to keep the old version around to provide users access to the version they might need for specific tasks.

In order to solve this problem, we observe that a file system can be viewed as individual files that are composed together into a single view. While some files will generally be grouped together, because they are part of a software package, other files will not be, because they are part of different versions of the same software package or because they are totally unrelated. A third category is files that are not generally grouped together, but depend on each other. For example, software packages can depends on system libraries to run correctly. This simplified view of how package management systems work, is based on the premise that their primary tasks are to ensure the dependencies a package requires are met when one wants to install it, as well as extracting a set of files that belong together and placing them on the file system. The file system ends up with a composited view of the packages you install via the package manager. However, package management has traditionally viewed each machine as being distinct, which means, that

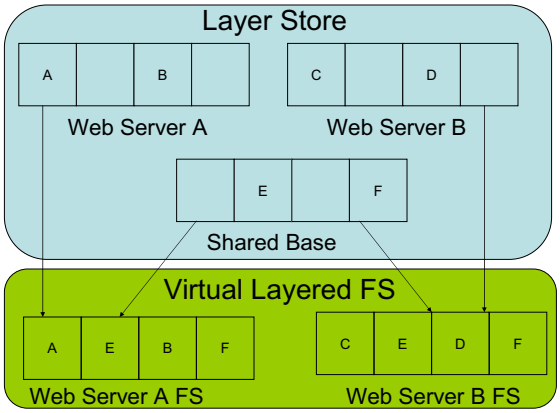

Figure 1: Simple Example of File System Layers

if one needs to upgrade a system library, for instance, due to a security hole, then the package manager must run on each individual machine to install the updated package into the file system.

Our solution is to view files that are generally grouped together as a layer. Just like a traditional file system provides a view that can be viewed as a composite of individual files, Strata's file system provides a view that is a composite of individual file system layers. Just like an individual software package might depend on other software packages being available for it to work correctly, so to layers can explicitly depend on software provided by other layers in order to work correctly.

By viewing a file system as a set of layers that can be composed, we solve all of the problems described above. First, by storing the layers on a file server, the layers are readily available to large number of machines that want to use them. Second, by utilizing layers, we enable easy provisioning, due to the fact that layers are readily reusable since each layer can be shared in a read-only manner. In fact, in general use, each file system would be differentiated by treating the software layers as read-only while providing it with its own private writable layer that would be initially be blank. Finally, layers also enable us to solve easing upgrades. Since each system only makes use of layers, all one have to do is create a new layer that serves as the upgrade to replace the old layer. This enables each system to easily pull in the upgraded software by just using the new layer instead of the old layer. This also means that upgrades do not remove old software since upgrades are creations of new layers, not overwriting of old software as is the traditional way of upgrading software. This enables easy rollback of upgrades due to the fact that all one has to do is replace the layer again.

Instead of creating virtual appliances, one creates a virtual layered file system. Like a virtual appliance, virtual

layered file systems are able to share System state, in this case they share system layers, while maintaining a separate Data area. However, unlike virtual appliances, the Data area does not have to be distinct from the System area, as it is just another layer that is private to a specific virtual layered file system instance.

Strata enables the creation of virtual layered file systems as shown in Figure 1. Figure 1 describes a simplified use of Strata's layers, where there are two layers that provide different web servers, as well as a single layer that provides the rest of the file system. In a real world example, the *shared base* layer will actually be a set of layers, containing various system libraries and applications in their respective layers. Figure 1 demonstrates how Strata composes these multiple layers together into a fully composed virtual layered file system that can be used by regular applications.

This paper describes how Strata enables new ways to manage systems through the use of our virtual layer file system. Section 2 describes Strata's architecture. Section 3 describes the life cycle of a virtual layered file system. Section 4 describes in depth how the virtual layered file system can be used to solve many different types of problems. Section 5 describes how a layered virtual layered file system is implemented, while section 6 analyzes the performance implications of using a virtual layered file system. Section 7 discusses related work. Finally, we present some concluding remarks.

2 Strata Architecture

Strata's virtual layered file system provides the ability to quickly provision services. Just like virtual appliances, virtual layered file systems can be easily upgraded. Similarly, while virtual appliances can deal with rollbacks in a minimal way, virtual layered file systems enable one to rollback a single layer independently, enabling users to mix and match layers that fit their needs, not just rollback to a point in time.

The virtual layered file system goes a step beyond virtual appliances, since it can also be used to manage physical machines as well. While virtual appliances need a host that is managed independently to run on. Some virtual appliances provide tools for managing the host, but one ends up in a bifurcated world where one set of tools is used to manage the host, while another is used to manage the virtual appliances that are run on that host. Virtual layered file systems provide a single way of doing both

Like virtual appliances, Strata enables server consolidation by allowing multiple virtual layered file systems to be in use on a single machine. Unlike virtual appliances, virtual layered file systems do not restrict the type of environment one needs to use, but one can use virtual layered file systems in whatever level of isolation one wants, be it a *chroot* environment, a virtual server environment, virtual machine or even a physical machine. This provides an administrator with greater control over one's physical resources due to not having to waste resources on high overhead solutions when lower overhead ones are sufficient. By consolidating multiple machines into distinct virtual layered file systems running on a single server, one improves manageability by limiting the number of physical hardware and the number of operating system instances an administrator has to manage.

Traditionally, an administrator creates a monolithic file system for each installation, such as by installing all the software packages requires on each individual machine. One way some systems enable administrators to improve this situation is to take an existing installed system and create independent clones. However, this results in each independent clone having to be upgraded separately when a package is upgraded, for example, due to a security hole being fixed. Virtual appliances improve the situation due to each cloned appliance being able to pull in the updates from the parent it was cloned from. However, one would still have to upgrade each distinct virtual appliance independently. Strata's virtual layered file system changes the model by creating virtual layered file systems composed of shared software layers. Each software layer can contain self contained applications, such as a web browser, photo manipulation program, or system libraries. These layers are analogous to software packages an administrator installs into a traditional file system. Just like installing multiple packages into a single file system composes the packages files together, so too layers can be composed together based on the requirements of each machine. Unlike traditional systems, since software layers can be shared, if a program or library, such as the traditional C library, has a security hole discovered, all an administrator has to do is create a new layer that contains that fixed software and it can be automatically pulled in by all the systems that use the layer.

While independent virtual layered file systems can share software layers in a read-only manner, they are made independent by providing each virtual layered file system with its own private read-write layer. This private layer provides a place where any created or modified files are written. This enables virtual layered file systems to be differentiated, as any changes are private to it, as well as isolating the shared software layers from any changes that occur. While the same file can exist in multiple layers, only the top most version of it will be visible. If the file is modified, it will be written to the top most private layer of the virtual layered file system, concealing the copy that exists on the software layer below. Therefore, if a software layer, which can contain default versions of configuration files, is upgraded, administrators do not have to worry about it overwriting their configuration changes, as no files are overwritten and their modified copy will conceal the version below. Alternatively, if an administrator is happy with the default configuration provided by the layer, one does not have to

change anything as it will be composed into the file system correctly. This also enables administrators to easily identify the files that have changes in a specific virtual layered file system, since all the changes will be isolated to the virtual layered file system's private layer.

Software layers are composed of three things, files, configuration scripts and meta data. First, files are simply the individual items in the layer that can be composed into a larger virtual layered file system. There is no restriction on the type of files, they can be regular files, symbolic links, hard links or device nodes. Second, configuration scripts are the scripts that have to be run when a layer is added or removed from a virtual layered file system, to enable proper integration of the layer. While many layers are just a collection of files, other layers need to be integrated into the system as a whole. For example, a layer that provides `mp3` file playing capability would want to register itself with the system's MIME database to enable programs to launch it automatically when they want to play the `mp3` file. Similarly, if the layer would ever be removed, the system should remove it from the MIME database. Finally, a layer contains meta data that defines its name, what version it is, and most importantly dependency information. This dependency information is important in order to ensure that virtual layered file systems are composed correctly. Many layers will depend on the presence of other layers, notably system libraries, in order for them to work correctly. The dependency meta data enables layers to depend on specific layers, as well as specific versions of the layers.

A virtual layered file system is defined by which explicit layers one wants within the file system view. These can be either simple layer names, which imply the most recent version of the layer, as well as being able to specify an explicit version to be used. These explicitly specified layers in turn implicitly pull in the most recent version of any layer that satisfies their dependencies. When a virtual layered file system is upgraded, Strata checks for the most recent version of all explicitly specified layers if a version was not initially specified, as well as the most recent version of layers that satisfy the dependency graph created by all the layers. In this way, if a layer is upgraded, for instance, the standard C library due to a security hole, the virtual layered file system can be easily upgraded because Strata will notice there is a newer version of the layer that belongs to the file system and that it satisfies the dependency graph.

Strata provides two ways for an administrator to access layers. First, an administrator can use a publicly managed layer repository, much like one would use a publicly managed package repository from a regular Linux distribution. Instead of downloading and installing individual packages into a traditional file system, the administrator simply mirrors the layers one wants to be available locally by downloading and extracting the individual layer archives one wants into the personal layer store. Once layers are available within the personal layer store, they can be used to form a virtual layered file system.

Strata also provides the ability for administrators to create their own layers, for software that does not exist within the publicly managed layer repositories. While the administrator could install the software into the private layer of an existing virtual layered file system, it would not be available to any other virtual layered file system, and would have to be installed manually on each one. Therefore, instead of installing software into the current file system directly, the administrator only adds a new blank layer into the current file system. The administrator can then perform the installation in a regular fashion and the new layer will contain the newly installed software. By installing the software into its own layer, the software is isolated from the rest of the system. Once a layer is created, the administrator will tag it with a name and a version. In order for layers to be composed correctly, Strata also extracts the important meta data from each layer, such as what libraries and programs it provides and which ones it depends on.

In order for the layers to be available for in every instance where they will be used, they can be stored on a centrally managed file server. This provides reduced management, due to only needing one server to be managed and backed up. As a single server could provide a bottleneck, especially in view that Strata is meant to scale and support many individualized instances, the Strata file system caches the layers locally on the machine they will be used on. While file system writes will still have to propagate to the server, for most applications where file system reads dominate, such as web servers or a user's desktop this lessens the load on the server. In case of more extreme load, a distributed or mirrored architecture can be used instead.

For example, to provide a web server, an administrator would select the Apache layer to be composed into the virtual layered file system. This layer only directly depends on 8 other layers, such as providing the ability to identify file mime types correctly as well as Perl that is needed by the scripts it provides. However, these layers in turn pull in a total of 40 layers. Strata determines all the layers that are needed automatically, creates a private layer for this virtual layered file system providing a total of 42 layers that compose this virtual layered file system. An administrator can create as many of these Apache virtual layered file system as needed, and the only space that will be allocated is to hold the private layer of each individual virtual layered file system. If a bug is discovered in any of the layers, the administrator simply extracts or creates a new layer that fixes the bug and each virtual layered file system will be upgraded the next time they are reconfigured.

3 Strata Life Cycle

Strata manages the life cycle of virtual layered file systems by isolating distinct states the file system exists in. This creates a state model that enables administrators to eas-

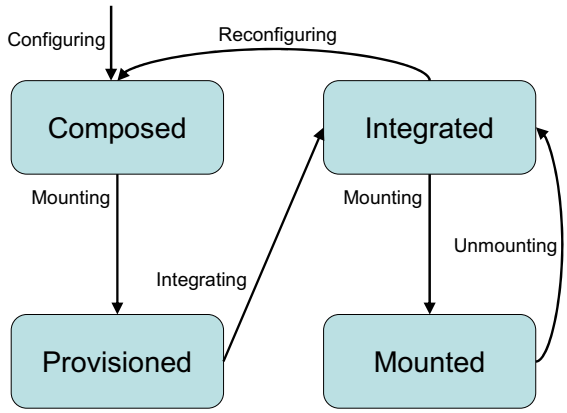

Figure 2: The virtualized FS life cycle

ily understand what actions are supposed to occur at what times. Figure 2 depicts Strata's state model. Strata enables the file system to exist in 4 distinct states, while also allowing 5 types of state transitions.

The four states are:

- **Provisioned**: An administrator has selected which software layers should be used by this virtual layered file system, as well as allocated a private layer for use by this file system
- **Composed**: The virtual layered file system's layers are fully composed and the file system is mounted and accessible to the system. However, the layer configuration scripts have not run yet. This is analogous to extracting a file archive into an existing file system. This is an intermediate stage in configuring the file system properly.
- **Integrated**: The virtual layered file system has run all the layer configuration scripts, and is able to be mounted and used by the system.
- **Mounted**: The virtual layered file system's layers have been integrated properly and composed together on a mount point making the file system accessible and fully usable

The five transitions are:

- **Configure**: An administrator selects which layers should belong to a specific virtual layered file system. For instance, if an administrator wants to run a mail server, one would explicitly select a layer containing a program such as Exim or Sendmail. This in turn would automatically implicitly pull in the appropriate layers that are needed in support of the specific mail transfer agent.
- **Reconfigure**: If Strata detects that the administrator of the virtual layered file system has changed the configuration, for example, by adding or removing a layer from the configuration, or that one of the explicit or implicitly defined layers needs to be upgraded, the file system will automatically reconfigure itself. This is needed, because any time a layer is added or removed from the virtual layered file system, the old layers will have to be disintegrated from the virtual layered file system, while the new layers will then be integrated.
- **Integrating**: Strata runs the installation and removal scripts that belong to the layers that are being added or removed from this virtual layered file system instance. Strata is able to perform common actions, such as running `ldconfig` to setup the appropriate shared library symbolic links, automatically.
- **Mounting**: Strata composes the file system out of its component layers and makes it available to the system to be used for either integration purposes or regular running purposes.
- **Unmounting**: Strata decomposes the virtual layered file system into its requisite layers, and it is unavailable to be used. While the file system is in this state, the system is guaranteed that the file system will not be modified and is therefore able to get a consistent backup if necessary.

If the traditional root file system of a machine is being replaced by a layered file system, Strata provides the ability to manage these states directly without administrator intervention. While an administrator can (re)configure the explicit layers while the file system is on/off-line, once the machine (re)boots, Strata will automatically determine the full configuration by implicitly including the appropriate layers. If this virtual layered file system has layers being added or removed from it, for instance, due to it being a new file system or being reconfigured, it will compose the file system in order for it to be integrated and finally mount it. If it is already been integrated, it will automatically mount it. Before Strata mounts an already existing virtual layered file system, it will determine if any of its component layers need to be upgraded and automatically reconfigure it if necessary.

4 Examples Usages

We briefly describe four scenarios that illustrate how Strata can be used to improve the ability of administrators to manage large numbers of systems. We first describe how one would use Strata to manage virtual machines that contain disparate applications. We then describe how one would manage the physical machines these virtual machines run on. Third, we also describe how an administrator can allow

his users to run different versions of an application. Finally, we describe how Strata can enable an administrator to better handle machines compromised by an attacker.

4.1 Managing Multiple Virtual Machines

Administrators like to run many services on a single machine. By doing this, they are able to benefit from improved machine utilization, but at the same time give each service access to many resources they do not need to perform their job. A classic example of this is e-mail delivery. E-mail delivery services, such as Exim, are often run on the same system as other Internet services to improve resource utilization and simplify system administration through server consolidation. However, services such as Exim have been easily exploited by the fact that they have access to system resources, such as a shell program, that they do not need to perform their job. Similarly, an administrator might want to run a web service, such as Apache. However, Apache itself has also been exploited in the past [11] and has been compromised many times due to insecure web applications [18] and misconfigured application frameworks, such as various PHP exploits. [21].

In order to solve these problems, an administrator would prefer to run each service in its own isolated virtual machine. Even if the service would be exploited, the attacker would only have access to the resources of the virtual machine which can be limited to just what the service needs to use. For example, Strata can run both Exim and Apache within their own personal virtual machines that are restricted to just the programs they need to execute. However, by placing each service within its own virtual machine, one can double the workload needed to maintain these services. Since each virtual machine needs its own file system, if a security hole in a shared file, such as the system C library, is discovered, the administrator has to upgrade each virtual machine independently. As administrators deploy more application and service specific virtual machines, this increased load will increase linearly.

Strata enables an administrator to have the benefits of deploying many virtual machines, while also not requiring an ever increasing burden on system administrators. On a simplistic level, Strata enables each virtual machine to have a virtual layered file system that is composed of 3 sets of layers. The first set of layers can contain all the files that are shared between the two services, while the second set of layers will be the files, such as those that belong to Exim and Apache that differentiate each service from each other. Finally, a third layer is provided to each independent virtual layered file system that remains private to each virtual layered file system instance, to enable the file system to be writable. This enables multiple Exim and Apache virtual layered file systems to exist in parallel as the third layer will contain each file system's personal modifications.

By splitting the file system into layers, sharable and private, Strata enables administrators to easily upgrade all the virtual machines they manage with a single upgrade action. For instance, if a security hole in the libc system library was discovered [4], all the administrator would have to do is create a new layer that contains the security update. The old layer is still accessible, but will not be chosen unless explicitly asked for. Once the virtual machines reboot themselves, they will automatically reconfigure their virtual layered file system using the updated layer.

4.2 Managing Multiple Physical Machines

While Strata enables one to manage myriad of virtual machines in an efficient manner, the virtual machines run on a physical host that also has to be administrated. Physical machine administration suffers from some of the same problems as virtual machine administration as well as introduces some of its own. First, just like virtual machines have a file system that has to be maintained by the administrator, physical machine's file system also has to be maintained. When security updates are available, the physical machine has to make use of them. Secondly, physical machines can suffer hardware faults. If a hardware fault propagates into the system, it can corrupt applications running within the virtual machines. Similarly, some software updates, such as newly installed kernels, can only take effect once the physical machine has been rebooted. However, by rebooting the machine, all the virtual machines on the host will be rebooted as well, which will force all the virtual machines to be shutdown as well, possibly causing them to lose their computation state.

Strata enables us to solve these problems by leveraging its ability to manage file systems and combining it with a system, such as Zap [12], that provides the ability to checkpoint, migrate and restart applications that are run within a pod. Strata manages a physical machine's file system in the same manner that it manages the virtual machines it hosts. First, it is able to manage the host machine's file system with layers, and second it is able to support multiple individual pods with their own layered file system. Therefore, Strata provides the physical machine with a set of layers that contains the Strata software and its dependencies, as well as a private customization layer for it to store the machine specific data, such as logs and configuration files. When Strata has to be upgraded, for instance due to a security hole or new features, Strata can simply reconfigure itself to form its virtual layered file system with the new layer that contains the updated software and its private customization layer. Since all the applications are run within pods, none of their state will be lost, since they can simply be checkpointed and restarted.

Strata can also be combined with AutoPod [15] to monitor the physical system for hardware faults. While the virtual machines that run on the physical host do not have hardware that can fault, they depend on the physical machine. Faults on the physical machine will propagate into

the virtual machines and corrupt execution within it. AutoPod therefore monitors the physical machine state, and if it discovers imminent failures it can checkpoint the pods that are running on it to ensure they do not lose state. It can choose to migrate the pods to a new physical machine, if one is available, so that they can continue running as if nothing has happened. Since the Strata stores the file system layers on a centrally available server, the file system layers are composable on other machines. Even if there is no machine to migrate too, once the problem is repaired, the pods can be restarted exactly where they left off, causing no loss of computation to users.

4.3 Managing Multiple Software Versions

Software packages that are in used today are very complicated, with varying types of configuration systems and numerous options that can be set. These software packages are generally part of a larger system of different software packages that work together to form an even more complicated system. However, these software systems have to be maintained and upgraded. Many times a new version of the software will be released that contains new features that users want access to. Similarly, software has to be upgraded many times due to bugs that can impact users, or security holes that can impact the entire system.

A main problem administrators face in upgrading software, is that they are moving their systems from a known state, where they know that the software works, to an unknown state. Even if an administrator has separate systems to test out the new software on, many times the software still breaks when placed in the real world. However, rolling back the software to the previous version is not always easy, as the older version might not be readily available, and the upgrade overwrite the original version. Even if one has a backup available, the administrator would have to extract the upgraded software from the backup image which is a time consuming and difficult task. While software packages improve this situation, many times a software package depends on a specific set of dependencies that are no longer available within the running system and therefore can not be installed.

Strata improves this situation in three ways. First, Strata enables administrators to easily provision test virtual machines. Second, if the upgrade happens and is shown to not work in real life, Strata enables an easy rollback. Third, even if the older package can not be installed into a current system due to dependency conflicts, Strata enables quick provisioning of a new system with the required dependencies. Strata enables easy testing and rollback through its virtual layered file system and use of layers. First, an administrator would create a new layer with the upgraded software. The administrator would then provision a new virtual layered file system to use that layer instead of the older software used in the original virtual machine. The administrator could then run his tests against the upgraded software in the new virtual machine. If the administrator is satisfied with the results, he can then instruct the original virtual machine to begin using the updated software layer. However, if, during the course of real world use, it is discovered that the upgraded software breaks, an administrator can easily rollback his virtual machine to the older version by simply using the older layer. Since Strata never throws away older software versions, rollbacks are simple.

A similar problem is faced by some users and administrators when software has to be upgraded due to new features users want or security holes. Many times upgrades software will be known to not be 100% backwards compatible, but has to be upgraded anyways. Strata administrators are able to provide their users with the new software by default, while enabling their users to create other virtual machines that contain the older software as well. For example, while LaTeX 2_ε is mostly backwards compatible with LaTeX 2.09, it is not fully. Therefore, an administrator can either choose to figure out how to have both versions of LaTeX installed in parallel or can go the much simpler route of creating a LaTeX 2.09 layer that can be used in a small virtual machine for the few users who need to rebuild old documents. This also enables users and administrators to use software packages with known security holes in limited VMs that are isolated from other processes executing on the same physical host.

4.4 Fixing Compromised Machines

Machines face attack on a continuous basis and while administrator work to prevent the attacks from succeeding, the attacks do succeed from time to time. One of the main problems administrators face in dealing with a compromised machines is that one does not always know what the attacker modified. Therefore, the best course of action, many times, is to completely reinstall the machine from scratch. This results in two problems. First, reinstalling a system from scratch can take a long time. Second, reinstalling a system from scratch can make one lose all the data that belonged to the system. While an administrator can backup the system before its reinstalled, this further adds to he time it takes to restore the compromised system.

Strata changes this scenario in two fundamental ways. First, by leveraging virtual layered file systems, it becomes simple matter of recreating a fresh system. Since Strata creates virtual layered file systems out of a set of shared read-only layers composed together with a read-write layer, even if an attacker is able to compromise the system, he can not compromise the underlying data layers. In order to ensure this, the read-only nature of the shared layers is not enforced locally, by the machine hosting the virtual layered file system, but by the remote server. By storing the software layers on a remote distributed file system server that enforces the read-only semantics of the share the layers are on, an attacker who breaks into a virtual machine w/ a virtual layered file system will not be able to modify

the underlying shared system data that makes up the virtual layered file system without breaking into the system that stores the layers. An attacker will be able to compromise this instance of the virtual layered file system, since when he modifies files on it, they will be copied up to the private read-write layer of the file system. However, this provides three benefits. First, any change the attacker made will be clearly visible by examining the private layer. Second, an administrator can instantly clean the system by replacing the compromised private layer with a fresh layer. Finally, since cleaning the system does not require ridding oneself of the compromised private layer, an administrator does not have to waste time backing it up and can also make it available within the virtual layered file system as a regular directory without it being composed into the normal file system view.

While quickly fixing compromised systems is useful, this often results in throwing away the configuration changes an administrator has made for that system. In all the above cases, we described a single virtual layered file system that contained multiple read-only layers that make up the System and one read-write layer that contains the Data areas. However, the Data area does not have to be limited to a single layer. For instance, an administrator can create a virtual layered file system as described above that only has one read-write layer for the Data area. Any configuration changes the administrator of the file system makes, will be confined to this read-write layer. Generally, the configuration changes an administrator makes are done up front and remain static for an extended period of time. Therefore, one can also add another layer to the private Data area. An administrator would first uses the virtual layered file system as described above, with a single layer for the private Data area, and make the appropriate configuration changes. When the administrator is satisfied with the configuration changes, one adds a new private read-write Data layer while making the current layer read-only, resulting in the Data area being defined by two layers, one read-only and one-read-write. This locks down the changes the administrator made into the read-only layer, while still enabling them to be modified in the future. If the changes have to be blown away, due to fixing the virtual layered file system due to the system being compromised or simply due to an administrator corrupting the configuration in an attempt to modify it, one can simply revert back to the locked down configuration due to it being kept on a read only layer.

5 Strata Implementation

Strata's FS work by leveraging and expanding upon unioning file systems. Unioning file systems allow the system to join multiple distinct directories into a single directory view. These directories are unioned by layering directories on top of one another. For example, if one had 2 directories unioned together, one directory containing the file `foo` and the other containing the file `bar`, the unioned directory view would contain both files `foo` and `bar`. In order to provide a consistent semantic, most union file systems only allow one layer, namely the topmost to have files added to it. At the same time, if one uses a file that already exists, the unioning file system cause a user's file system modifications to change the underlying file directly, in whatever layer of the union it existed previously.

On the other hand, some union file system can also assign properties to the directory layers, defining some layers to be read only, while others can be read-write. This results in a model that borrows from copy-on-write (COW) file systems, where a modifying a file on a lower read-only layer will cause it to be copied to the topmost writable layer in a COW fashion. For instance, in the above example, the layer containing `foo` can be layered on top of a read only layer containing `bar`. If in the course of usage file `bar` get modified it will be copied to the top most layer, which will now contain both files `foo` and `bar`, before the modification takes place. However, since the entire file has to be copied, performance can suffer if this operation has to occur often on large files.

This layering model also provides a semantic that directories located at higher layers in the stack can obscure items at a lower level. Continuing the previous example, both layers now contain the file `bar`, but only the top most layer's version of the file is visible. In order to provide a consistent semantic, if a file is deleted, a "white out" mark is also created on the top most layer to ensure that files existing on a lower layer are not revealed. To continue the example, if the file `bar` were deleted, it would not allow the `bar` on the lower layer to be revealed. The white out mechanism also allows one to obscure files on the read only lower layers, by just creating the obscuring white out file on the topmost layer.

This union file system semantic provides a good base for supporting Strata's layering file system. Strata's layering file system provides the ability for multiple independent layers to be combined together into independent file system views, such that a file system can be composed of multiple software layers, as well as a private layer for each view. Each view can then be treated like a regular file system in that each can be actively modified state at same time since each view's modifications are confined to its personal layer. To provide a simple example, imagine one has a directory that one wants to branch into two distinct views. This implies that processes operating in one view would be able to modify the files, without the changes causing any effect in the other view, and vice versa.

This model is implemented by Strata with the above unioning file system semantic. By using the union file system semantic described above and forcing all lower layers to be read only and maintaining a single top most writable layer, Strata can simply create two distinct views of the di-

rectory by creating 2 distinct file system view, where Strata layers the common directory structure in each view with a blank directory stacked on top of it. Since all modifications will cause files to be copied to the top most directory, it enables one to simply contain each views modifications into its own space.

While most administrator will access layers through a centrally managed layer repository, such as one is used to using with packages, Strata provide the ability for administrators to create their own layers. Creating a software layer leverages the skills system administrators already have, namely the ability to install software. In order to make it easier on system administrators, Strata leverages the ability to use the regular tools they are used to using, namely package management tools such as the Debian package management tools `dpkg` and `apt`. The general way an administrator would install a set of software packages is by simply telling the package manager to install those packages. Fundamentally, what these package management tools do is ensure that the packages can be installed, unpack the software packages and update its internal database with the fact that these packages were installed. Strata changes this in a single way. Instead of installing the software packages into a monolithic file system, an administrator creates a new blank layer on top of the file system, and then runs the package management tools. This captures the software installation into the new layer which can then be composed with other layers instead of being just contained within a single file system.

In order for layers to be composed together and work correctly, Strata has to provide functionality similar to a package manager. Just like a package that a package manager installs can depend on other software being available on the machine, a software layer in Strata can depend on software other layers contain. Strata is able to extract dependencies in two ways. First, for software installed via a package manager, it is able to extract the dependency information provided by the packages. For example, packages can depend on another package being installed or even a specific version of that package. Packages can also provide virtual package names, as multiple software packages might depend on specific functionality, such as a local mail or web server, but does not care which software package one uses to fulfill that functionality. Strata extracts this information and divides into three categories, packages, provides and depends. Packages are the individual packages installed into a layer, Provides are the packages that are provided plus the virtual packages that they might provide, while Depends are the software packages that the packages in the layer depend on. Strata also supports the ability to create a software layer by hand, if an administrator would build it from source code. In this case, Strata automates the ability to extract certain dependencies, such as libraries the processes will require to dynamically link, but cannot automate hidden dependencies, such as services that might be required, and would have to be specified by the administrator manually. Strata does not just provide simple package names, but enables the layer's dependencies meta to be versioned and for matching on only specific versions. In specific, Strata provides 6 operators that can restrict which versions will fulfill a dependency, namely 1) less than, 2) less than or equal, 3) equal, 4) not equal, 5) greater than or equal, 6) greater than. In fact, a layer can depend on the same package with more than one restriction if it can be only used with an internal subset of versions.

Strata resolves dependencies in a matter similar to how modern Linux distributions resolve package dependency and automatic installation of required packages. Strata has knowledge of all the layers available to it and the interdependencies between layers. Strata uses this information to create a large directed dependency graph with two types of nodes. The first type of node corresponds to each existing layer, while the second type of node corresponds to distinct dependency information. If layer A depends on layer B with version X, while layer B depends on layer B with version Y, this will create two distinct dependency nodes. Each existing layer has a set of directed edge to the dependency nodes that correspond to its dependencies, while each dependency node has directed edges to any layer that fulfills the condition set by that dependency. The graph is traversed to determine if the explicit layers can have their dependencies filled, and then recursively over the dependencies.

Strata's architecture is designed to allow efficient scaling and enable the layers to be used by many physical and virtual machines. Strata accomplishes this by coupling a network file system distribution approach with local caches on each physical machine the layers will be used on. Strata uses NFS to distribute the layers to many machines. NFS is standardized and readily available on commodity machines and well understood by system administrators. In addition, Strata combines the NFS file system that stores the layers with a persistent file system cache, namely it survives reboots, that enables Strata to treat many file system reads as if they were local reads, instead of remote reads. The file system cache is configurable in size, as well as storable on portable media, which enables the cache to move between computers if necessary.

In order for Strata to use the cache efficiently, the cache is designed to only cache data on a page by page basis, as that is the same size the underlying operating system will read data from the file system. This enables two important cache properties. First, Strata is able to cache files, even if their total size would be larger than the cache itself. Second, on an individual file level, Strata is able to cache files that are larger than the cache itself. In order to prevent temporary usage patterns from evicting important data from the cache, Strata also supports the ability to pin data into the cache. Data is removed from the cache when the free spaces drop below an administrator configurable level,

based on the least recently used algorithm that takes into account when the files storing the individual cache object's access time was updated by the kernel.

By using a cache, Strata lowers the load on the centrally managed file server. However, depending on scalability, Strata can overwhelm a single file server. Strata therefore takes advantage of the fact that its software layers are static; once they are created they are not modified directly, which enables them to be easily mirrored across a set of machines that can be used to scale as load increases. The problem with this approach is that each Strata file system view can have its own private layer that must be stored somewhere. If its stored on a file server, it wont be available if it must use a new file server until the layer is able to be copied to it. In order to solve this problem, Strata allows the private layer to be stored locally on the machine where that file system view is being used. If the private layer is being stored locally, existing backup solutions can be used to ensure that recovery in case of disaster. On the flip side, this approach prevents a virtual machine from being migrated to another host. Therefore, Strata supports storing the private layer on both the layer server, as well as locally depending on the needs of the users.

5.1 Setting Up and Using Strata

To demonstrate how one sets up a layered file system and makes use of Strata, we provide a step by step walk-through on how one composes layers together into a file system, as well as manually upgrades them when a new layer becomes available. While one would generally use a centrally managed layer store and not have to create one's own layers, we demonstrate layer creation via conversion of a Debian GNU/Linux packages into layers. We demonstrate this by walking through how an administrator would setup and use file systems layers for implementing our example of multiple independent services of Apache instances with their own independent file systems. Setting up layers is a straightforward task and leverages many of the same skills and experiences system administrators already have on a standard Linux system.

While the majority of administrators would download layers from a centrally managed repository, our prototype Strata system enables an administrator to convert a Debian GNU/Linux binary package into a self contained layer. Debian binary packages are generally composed of three distinct items. First, they are composed of files that would be extracted into a given file system. Second, they are composed of configuration files that belong within the file system, but can be changed by the administrator and therefore should not always be overwritten. Finally, they are composed of metadata. Strata takes the set of packages one wants within the layer and extracts both the regular and configuration files into a blank directory that will now contain the layer. Then, Strata extracts the meta data from the packages, including the package names and versions as well as the dependency information. Finally, Strata insert the meta data information into its own database so that the layers can be used as part of a virtual layered file system. All an administrator does is download the appropriate Debian binary packages and runs `strata-create-layer <packages>` in order to create the layer. In order to create a simple apache layer, an administrator would download the apache binary packages one wants within the layer into a blank directory and run `strata-create-layer *.deb`. This takes all the apache binary packages one downloaded to the current directory and makes a single layer out of them. Once a layer is created it can be simply stored as a regular `tar` archive and copied between systems. Strata provides `strata-archive-layer` and `strata-extract-layer` commands that automate archiving and inserting layers into a new layer store.

Once an administrator has a layer store available to him, all one has to do is make use of it to create new virtual layered file systems. In a standard Debian GNU/Linux system, an administrator would use the `apt-get` program to install a package and all of its dependencies into an existing file system. In Strata, An administrator can create a virtual layered file system and add layers to it in a similar manner. First an administrator creates the virtual layered file system by running `strata-create-fs <name>`. This creates the private read-write Data layer for it. Then an administrator adds the appropriate layers to it by calling `strata-add-fs-layers <name> <layer_names>`. The layers explicitly added with this command are the explicit layers that belong to the file system, while the layers that need to be used to provide all the requisite dependencies are the implicit layers. An administrator could also select specific versions of a layer by specifying `layer_name-layer_version`. To create a simple apache file system, we first create the file system by running `strata-create-fs apache1.3`. We then add the apache layer explicitly into the file system by running `strata-add-fs-layers apache1.3 apache`. This explicitly selects Apache, and implicitly pulls in another 40 layers that are needed because of dependencies, including items such as the Berkley DB library and Perl.

In order to demonstrate what occurs when a security hole occurs, we show how an administrator would deal with a real Debian security hole [5]. This security hole in Perl, enables a user to exploit a race condition in a directory tree being deleted by Perl's File::Path::rmtree function. In order to ensure that all systems are updated, an administrator would simply create a new layer containing the update. Once the layer is available within the store, an administrator can simply update it manually by running the `strata-update-fs <name>` command. This command takes an unmounted file system and reconfigures it if necessary. It does X items. First, it rebuilds a list of layers that belong to this file system based on the explicit layers

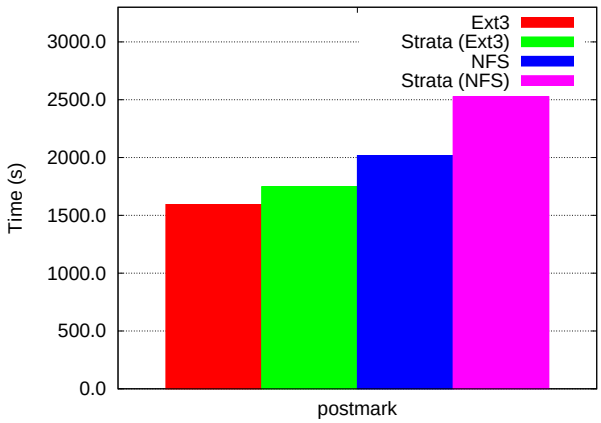

Figure 3: Postmark Overhead

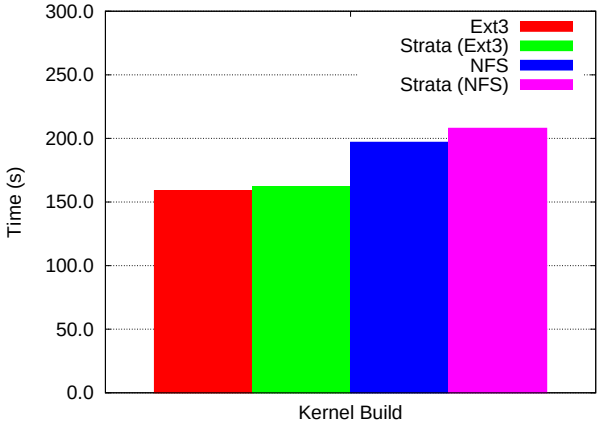

Figure 4: Kernel Build Overhead

assigned to it. Second, it compares this list against the list of layers and their versions already in use. If they have changed, Strata will remove and insert the appropriate layers from the existing file system. In our above Apache case, since it depends on Perl, it could suffer from the above security hole. Therefore an administrator maintaining this file system by hand would run `strata-update-fs apache1.3`. This would determine that a new Perl layer is available as it would have a higher version number, that it satisfies all the dependencies and would integrate it into the file system by removing the old layer and inserting the new one.

6 Experimental Results

We have implemented Strata as a loadable kernel module on the Linux 2.6.19 series kernel. Strata requires no changes to the Linux kernel, as well as a user space system status monitoring service. We present some experimental results using our Linux prototype to quantify the overhead of using Strata on various applications. Experiments were conducted on an IBM BladeCenter containing 14 IBM HS20 eServer blades with dual 3.06 GHz Intel Xeon CPUs and 2.5 GB RAM. The blades are interconnected by a gigabit Ethernet switch. To measure the virtualization cost of Strata's virtual layered file system, we used a range of micro benchmarks and real application workloads to measure the performance of our Linux Strata prototype and compared the results against vanilla Linux systems. The blade's local file system was formated with the `Ext3` file system, while it used the Network File System (NFS) over a 100 megabit LAN connection to talk to the remote file system. We tested Strata with its layers stored on both the local and remote file systems.

The first benchmark we performed was postmark [9]. Postmark is a synthetic test that measures how the system would behave if used as a mail server. Our postmark test operated on files between 512 and 10K bytes with an initial set of 20,000 and performed 200,000 transactions. Postmark is very intensive on a few specific file system operations, such as `lookup()`, as it is constantly creating, opening and removing files. As Figure 3 shows, while the difference between layers stored locally and the local file system is not minimal, the difference between the network file system and the network stores layers is much greater. Many file operations, such as `lookup()` require Strata to read the contents of all the underlying layers that provide files for the directory its working on. Strata is much quicker with a local file system due to the local file system being able to provide the directory contents much more efficiently than the remote NFS server can. While Postmark shows that Strata's virtual layered file system is not the best suited for workloads where files will be constantly created and removed, such as the mail server it mimics, we do not believe this is a major limitation. A user could still benefit from Strata by using it as its intended and providing a local vanilla file system to handle the workload of the constant creating and removing files. In fact, this is how many systems are setup already, with separate `/usr` and `/var` file systems. This allows `/usr` to remain relatively static, or even read-only, while `/var` can by very dynamic.

In order to demonstrate that Postmark's results are not indicative of real application performance, we performed two application benchmarks. The first benchmark was a multi-threaded build of the Linux 2.6.19.1 kernel with up to eight concurrent file compilations. Our second application benchmark placed a load on the Apache web server and measured the amount of connections that were completed per second. As can be seen from Figure 4, while Strata imposes a slight overhead on the kernel build compared to the underlying file system its using, it is relatively negligible at under 5% in the worst case. On the other hand, Figure 5 shows that when the `http_load` [14] program is used to fetch from a set of files averaging 1MB, it is able to saturate a 1Gbps Ethernet connection in every circumstance, even when using the remote file system over a 100 megabit ethernet connection. This is partly due to the fact that the machine running Apache in the different file sys-

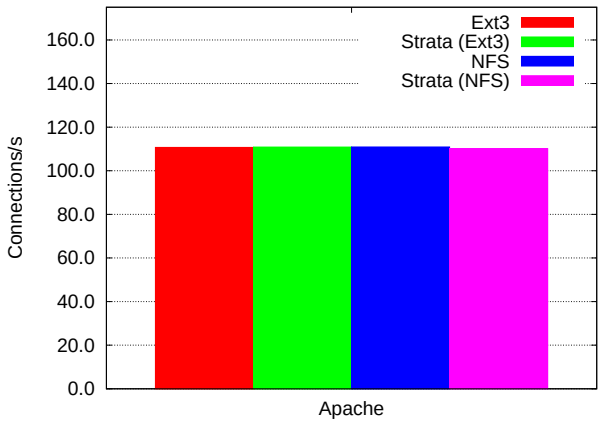

Figure 5: Apache Overhead

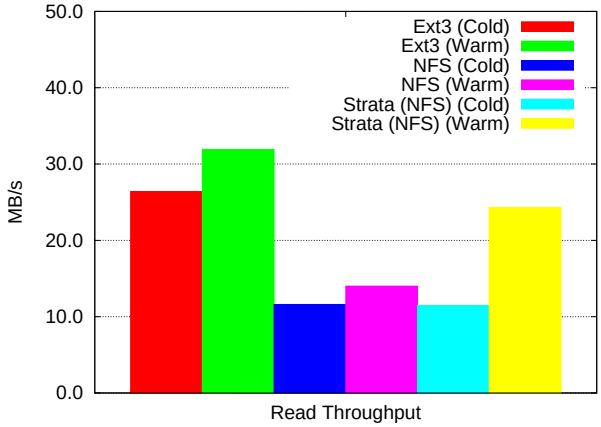

Figure 6: Read Throughput

tem environments was able to cache both the `lookup()` information as well as the file data.

In order to demonstrate the direct throughput advantage of caching, we measured how Strata impacts read and write throughput for a 3 gigabyte file. The throughput tests were measured by using the `dd` utility to write out a file of zeros, as well as read in the same file in 4KB blocks. We measured the throughput in multiple categories to quantify the impact the different elements of the Strata file system impact throughput performance. Figure 6 demonstrates that while network file system performance is predictably much worse than the local file system, when combined with Strata's local file system cache, performance approaches the regular local file system. While our current Strata prototype does not implement a write-back cache, and therefore cannot improve write performance Figure 7 demonstrates that Strata's layering does not impact performance negatively either.

In order to show how Strata helps administrators manage updates, we quantify the updates that went into Debian stable during 2005 and 2006. In 2005, Debian released 332 security updates, while in 2006, they released 345 updates.

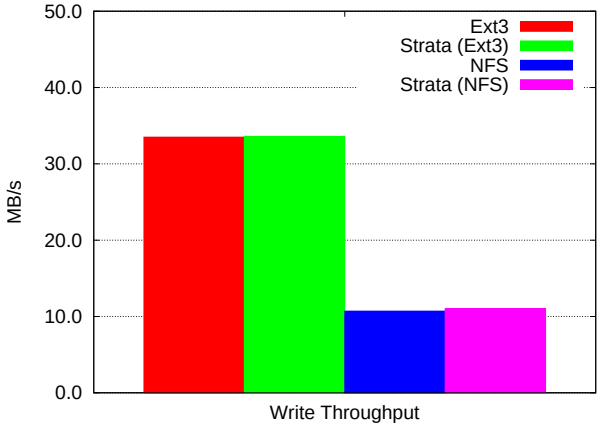

Figure 7: Write Throughput

Of these 677 updates, 53 of them were updates to previously updated packages, with 50 of the packages needing a second update, while 3 needed a third update. 389 individual packages had security holes that had to be fixed, with 280 only having one security hole that had to be fixed, 57 had two security holes, 26 had three, 16 packages had 4 security holes. 5 packages had 5, 3 packages had 6, while 2 packages had 7 fixes. Many of the packages with multiple required fixes are common packages one finds on Linux systems, such as squid and etheral with 7 security releases, xine-lib which forms the basis of many media players with 6, Squirrelmail and MySQL with 5 each. This shows that common applications that many people run will have to be updated multiple times in a year and that administrators need effective ways to manage these fixes.

In order to quantify how long it takes to setup Strata's virtual layered file system, we compared it against traditional methods. We created virtual layered file system containing an Apache server made up 43 layers, while we use the debootstrap and apt-get programs to provision a minimal apache installation in a Debian GNU/Linux Environment. Finally, we measure how long it would take to unpack a tar archive of the same installation. Strata is able to create the virtual layered file system containing Apache in .1 seconds. Comparatively, untar was able to extract a 194MB archive containing a full apache file system in 22 seconds, while `debootsrap` and `apt-get` took a combined total of 121 seconds, where `deboostrap` took 106 seconds to install a base Debian installation, while the `apt-get` took 15 seconds to download and install apache and its dependencies. In both of these cases, these tools downloaded from a local web server in order to minimize the impact of network latencies.

7 Related Work

Strata bears a resemblance to file system versioning and source code control systems. Operating systems such Tops-

20 OS [6] and VMS include native operating systems support for a versioning as a standard feature on their file systems. These OSs employ a copy on write semantic that involves versioning a file each time a process changed it. Other file systems, like VersionFS [10], ElephantFS [19] and CVFS [22] have been created to provide a better interface to the file system versioning metaphor.

Similarly, source code control systems, such as SCCS [16] and CVS [8] enable users to record the state of a file or set of files, while also enabling state to be branched and reverted. The ability of source code control systems to branch state bears a resemblance to Strata's shared layers being differentiated with a private layer. While the operating system, file system and source code control approaches could enable a system administrator to rollback state f a single machine if an upgrade occurred in error, they do not allow an administrator to compose a file system out of different entities or manage large number of machines efficiently.

Plan9's concept of namespaces and the ability of administrators to modify namespaces at will also bears a resemblance to Strata's use of layers. Plan9 viewed the entire operating system's set of resources as files, and therefore depending on the resources one wanted a process to access, one could provide processes with individualized namespaces. Strata, on the other hand, is focused not on limiting what a process can do, but on improving the ability of administrators to manage large numbers of machines.

The most common way of provisioning machines today is use of the package management system built into the operating system one is using, on Linux this would generally be Debian's Package Manager (dpkg) [7] or the Red Hat Package Manager (rpm) [17]. This approach closely mirrors Strata layering approach. While these systems work with individual packages that have to be installed on individual machines, each individual package is a set of files that belong together. Strata's layers are inspired by this concept. Strata however improves on this concept by enabling the set of files to only be installed once. If one has to manage a large number of machines via packages, ensuring that each machine gets updated takes a significant amount of time due to each machine having to be upgraded individually.

Both *The Collective* [20] and Ventana [2] attempts to solve a similar problem. The collective uses shared virtual disks to improve the ability of system administrators to manage large number of virtual machine instances. Each virtual machine in the collective will generally have 2 disks, a system disk that is shared and a private per machine virtual disk that contains content specific to the user or virtual machine. The collective combines these virtual disks with an intelligent caching architecture that enables a user to carry around a small cache device and have quick read/write access to their content, even in cases of limited access to the centralized servers that store the disks. Ventana is similar to Strata, but instead of providing sharable block devices, it provides sharable file systems. However, unlike Strata, both the Collective and Ventana manage the disks at either the block device level or the monolithic file system level, providing the user's with a single file system. While in an ideal setting administrators might be able to create one or very few shared images, in practice an administrator will need many images, and therefore will suffer from having to update each independent image. Strata on the other hand just composes a virtual layered file system together, so no matter how many virtual layered file systems a system administrator needs to support.

MIT's Project Athena [1] shares some similarities to Strata, in its goal to manage large numbers of machines. However, while its overall goals were much more broad, such as its ability to handle heterogeneous hardware, while providing users with a consistent environment no matter what machine they were running. Like Strata, Athena was centrally managed and distributed the majority of the software machines would run via network file systems. This enables an administrator to install software once and have it available on all systems. However, unlike Strata, in Athena all systems have to be consistent, and therefore does not let users or administrator customize a particular system that might be in use. While users can make use of some aspects of Athena, such as applications stored on centrally managed file systems, while still be managed independently, such as on a user's personal machine, this prevents that machine from receiving many of the benefits of centralized management.

Systems like Symantec's Ghost [23] and Radmind [3] let administrators centrally manage machines. Unlike Strata, these systems have to be totally under the control of the system administrators. While Ghost manages file system images and is therefore limited in the ways that all file system or block based schemes are, Radmind shares Strata's concept of having a set of layers that are placed one on top of another. Unlike Strata, Radmind requires that the software be installed on the host itself, which adds to the time the machine is unavailable.

Strata is based on the union file system concept, but expands it use to managing large number of machines. While unioning file systems have been used to union data together, they have not been used as a basis for managing single or multiple machines. Strata makes use of a modified UnionFS file system [24] that provides the ability to memory map files correctly, thereby enabling applications that depend on correct memory map semantics to runs correctly, such as the startup scripts in Debian GNU/Linux.

8 Conclusion and Future Work

Strata's virtual layered file system enables system administrators to manage the virtual and physical computers under their control in interesting new ways. Virtual layered file

system provide for simple and quick provisioning of file systems, while also providing the ability to upgrade and rollback all the different file systems one manages easily. Virtual layered file systems are not limited to virtual machines, but can be used in any environment one would make use of a file system, be it a virtual machine, a virtual private server, a simple *chroot* environment or even the physical machine they are hosted on. We have implemented Strata on Linux without requiring any operating system kernel changes, and have demonstrated how virtual layered file systems can be used in real life situations to improve the ability of system administrators to perform their jobs. Our measurements on real world applications demonstrate that Strata imposes little overhead.

Strata raises a number of interesting follow-up research questions. First, while our Strata prototype works on generic file systems, such as Ext3 and NFS, what would happen if the underlying backing store provided a better feature set for Strata to perform its actions. Some file systems [13] provide the ability to perform COW operations at the block level internally, for example, to support file system snapshots. If this COW functionality could be exposed to Strata, it could improve the ability of Strata to copy the file in cases where it has to perform its own COW operations. Second, while Strata explores the benefits of layers to provisioning and maintaining a file system, the question can layers be used to branch a file system state into two distinct entities and what benefits can that bring to the administration and testing of systems.

References

[1] J. M. Arfman and P. Roden. Project athena: Supporting distributed computing at mit. *IBM Systems Journal*, 31(3), 1992.

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

[3] W. D. Craig and P. M. McNeal. Radmind: The integration of filesystem integrity checking with filesystem management. In *Proceedings of the 17th Large Installation System Administration (LISA 2003) Conference*, pages 1–5, San Diego, CA, October 2003. Usenix.

[4] Debian Security Team. Dsa-636-1 glibc – insecure temporary files. `http://www.debian.org/security/2005/dsa-636`, January 2005.

[5] Debian Security Team. Dsa-696-1 perl – design flaw. `http://www.debian.org/security/2005/dsa-696`, March 2005.

[6] Digital Equipment Corporation. Tops-20 user's guide, January 1980.

[7] J. Fernandez-Sanguino. Debian gnu/linux faq - chapter 7 - the debian package management tools. `http://www.debian.org/doc/FAQ/ch-pkgtools.en.html`.

[8] F. S. Foundation. CVS - Concurrent Versions System. `http://www.nongnu.org/cvs/`.

[9] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR3022, Network Appliance, Inc., 2001.

[10] 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. USENIX Association.

[11] M. Murphy. Apache 2.0.3.7 - 2.0.45 apt exploit. `http://www.milw0rm.com/exploits/38`, 2003 June.

[12] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.

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

[14] J. Poskanzer. `http://www.acme.com/software/http_load/`.

[15] S. Potter and J. Nieh. Reducing downtime due to system maintenance and upgrades. In *Proceedings of the 19th Large Installation System Administration Conference (LISA 2005)*, pages 47–62, San Diego, CA, December 2005.

[16] M. J. Rochkind. The source code control system. *IEEE Transaction on Software Engineering*, 1(4):364–370, December 1975.

[17] rpm.org - the rpm package manager. `http://www.rpm.org/`.

[18] SANS Institute's Internet Storm Center. Reports of bots exploiting pmwiki and tikiwiki. `http://isc.sans.org/diary.php?storyid=1672`, September 2006.

[19] 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 *The 17th ACM Symposium on Operating Systems Principles (SOSP)*, December 1999.

[20] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[21] SecurityFocus. Php file upload global variable overwrite vulnerability. `http://www.securityfocus.com/bid/15250`, November 2006.

[22] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in a comprehensive versioning file system. In *2nd USENIX Conference on File and Storage Technologies*, March 2003.

[23] Symantec Corporation. Symantec ghost solution suite. `http://www.symantec.com/enterprise/products/overview.jsp?pcid=1025&pvid=865_1`.

[24] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.