Improving Virtual Appliance Management through Virtual Layered File Systems

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

Abstract

Managing many computers is difficult. Recent virtualization trends exacerbate this problem by making it easy to create and deploy multiple virtual appliances per physical machine, each of which can be configured with different applications and utilities. This results in a huge scaling problem for large organizations as management overhead grows linearly with the number of appliances.

To address this problem, we introduce Strata, a system that combines unioning file system and package management semantics to enable more efficient creation, provisioning and management of virtual appliances. Unlike traditional systems that depend on monolithic file systems, Strata uses a collection of individual sotware layers that are composed together into the Virtual Layered File System (VLFS) to provide the traditional file system view. Individual layers are maintained in a central repository and shared across all file systems that use them. Layer changes and upgrades only need to be done once in the repository and are then automatically propagated to all virtual appliances, resulting in management overhead independent of the number of appliances. Our Strata Linux prototype requires only a single loadable kernel module providing the VLFS support and doesn't require any application or source code level kernel modifications. Using this prototype, we demonstrate how Strata enables fast system provisioning, simplifies system maintenance and upgrades, speeds system recovery from security exploits, and incurs only modest performance overhead.

1 Introduction

A key problem organizations face is how to efficiently provision and maintain the large number of machines deployed throughout their organizations. This problem is exemplified by the growing adoption and use of virtual appliances (VAs). VAs are pre-built software bundles run inside virtual machines (VMs). Since VAs are often tailored to a specific application, these configurations can be smaller and simpler, potentially resulting in reduced resource requirements and more secure deployments.

While VAs simplify application deployment and decrease hardware costs, they can tremendously increase the human cost of administering these machines As VAs are cloned and modified, organizations that once had a few hardware machines to manage now find themselves juggling many more VAs with diverse system configurations and software installations.

This causes many management problems. First, as these VAs share a lot of common data, they are inefficient to store, as there are multiple copies of many common files. Second, by increasing the number of systems in use, we increase the number of systems needing security updates. Finally, machine sprawl, especially non actively maintained machines, can give attackers many places to hide as well as make attack detection more difficult. Instead of a single actively used machine, administrators now have to monitor many irregularly used machines.

Many approaches have been used to address these problems, including diskless clients [5], traditional package management systems [6, 1], copy-on-write disks [9], deduplication [16] and new VM storage formats [12, 4]. Unfortunately, they suffer from various drawbacks that limit their utility and effectiveness in practice. They either do not directly help with management, incur management overheads that grow linearly with the number of VAs, or require a homogenous configuration, eliminating the main advantages of VAs.

The fundamental problem with previous approaches is that they are based on a monolithic file system or block device. These file systems and block devices address their data at the block layer and are simply used as a storage entity. They have no direct concept of what the file system contains or how it is modified. However, managing VAs is essentially done by making changes to the file system. As a result, any upgrade or maintenance operation needs to be done to each VA independently, even when they all need the same maintenance.

We present Strata, a novel system that integrates file system unioning with package management semantics and uses the combination to solve VA management problems. Strata makes VA creation and provisioning fast. It automates the regular maintenance and upgrades that must be performed on provisioned VA instances. Finally,

it improves the ability to detect and recover from security exploits.

Strata achieves this by providing three architectural components: layers, layer repositories, and the Virtual Layered File System (VLFS). A layer is a set of files that are installed and upgraded as a unit. Layers are analogous to software packages in package management systems. Like software packages, a layer may require other layers to function correctly, just as applications often require various system libraries to run. Strata associates dependency information with each layer that defines relationships among distinct layers. Unlike software packages, which are installed into each VA's file system, layers can be shared directly among multiple VAs.

Layer repositories are used to store layers centrally within a virtualization infrastructure, enabling them to be shared among multiple VAs. Layers are updated and maintained in the layer repository. When a new version of an application becomes available, due to added features or a security patch, a new layer is added to the repository. Different versions of the same application may be available through different layers in the layer repository. The layer repository is typically stored in a shared storage infrastructure accessible by the VAs, such as an SAN. Storing layers on the SAN does not impact VA performance because an SAN is where a traditional VA's monolithic file system is stored.

The VLFS implements Strata's unioning mechanism and provides the file system for each VA. Like a traditional unioning file system, it is a collection of individual layers composed into a single view. It enables, a file system to be built out of many shared read-only layers while providing each file system with its own private read-write layer to contain all file system modifications that occur during runtime. In addition, it provides new semantics that enable unioning file systems to be used as the basis for package management type system. These include how layers get added and removed from the union structure as well as how the file system handles files deleted from a read-only layer.

Strata, by combining the unioning and package management semantics, provides a number of management benefits. First, Strata is able to create and provision VAs quickly and easily. By leveraging each layer's dependency information, Strata allows an administrator to quickly create template VAs by only needing to explicitly select the application and tool layers of interest. These template VAs can then be instantly provisioned by end users as no copying or on demand paging is needed to instantiate any file system as all the layers are accessed from the shared layer repository.

Second, Strata automates upgrades and maintenance of provisioned VAs. If a layer contains a bug to be fixed, the administrator only updates the template VA with a replacement layer containing the fix. This automatically informs all provisioned VAs to incorporate the updated layer into their VLFS's namespace view, thereby requiring the fix to only be done once no matter how many VAs are deployed. Unlike traditional VAs, who are updated by replacing an entire file system [12, 4], Strata does not need to be rebooted to have these changes take effect. Unlike package management, all VLFS changes are atomic as no time is spent deleting and copying files.

Finally, this semantic allows Strata to easily recover VAs in the presence of security exploits. The VLFS allows Strata to distinguish between files installed via its package manager, which are stored in a shared read-only layer, and the changes made over time, which are stored in the private read-write layer. If a VA is compromised, the modifications will be confined to the VLFS's private read-write layer, thereby making the changes easy to both identify and remove.

We have implemented a Strata Linux prototype without any application or source code operating system kernel changes and provide the VLFS as a loadable kernel module. We show that by combining traditional package management with file system unioning we provide powerful new functionality that can help automate many machine management tasks. We have used our prototype with VMware ESX virtualization infrastructure to create and manipulate a variety of desktop and server VAs to demonstrate its utility for system provisioning, system maintenance and upgrades, and system recovery. Our experimental results show that Strata can provision VAs in only a few seconds, can upgrade a farm of fifty VAs with several different configurations in less than two minutes, and has scalable storage requirements and modest file system performance overhead.

2 Related Work

The most common way to provision and maintain machines today is using the package management system built into the operating system [6, 1]. Package management provides a number of benefits. First, it divides the installable software into independent chunks called packages. When one wants to install a piece of software or upgrade an already installed piece of software, all one has to do is download and install that single item. Second, these packages can include dependency information that instructs the system about what other packages must be installed with this package. This enables tools [2, 10] to automatically determine the entire set of packages one needs to install when one wants to install a piece of software, making it significantly easier for an end-user to install software.

However, package managers view the file system as a simple container for files and not as a partner in the man-

agement of the machine. This causes them to suffer from a number of flaws in their management of large numbers of VAs. They are not space or time efficient, as each provisioned VA requires time-consuming copying of many megabytes or gigabytes into each VA's file system. These inefficiencies affect both provisioning and updating of a system as a lot of time is spent, downloading, extracting and installing the individual packages into the many independent VAs.

As the package manager does not work in partnership with the file system, the file system does not distinguish between a file installed from a package and a file modified or created in the course of usage. Specialized tools are needed to traverse the entire file system to determine if a file has been modified and therefore compromised. Finally, package management systems work in the context of a running system to modify the file system directly. These tools often cannot not work if the VA is suspended or turned off.

For local scenarios, the size and time efficiencies of provisioning a VA can be improved by utilizing copy-on-write (COW) disks, such as QEMU's QCOW2 [9] format. These enables VAs to be provisioned quickly, as little data has to be written to disk immediately due to the COW property. However, once provisioned, each COW copy is now fully independent from the original, is equivalent to a regular copy, and therefore suffers from all the same maintenance problems as a regular VA. Even if the original disk image is updated, the changes would be incompatible with the cloned COW images. This is because COW disks operate at the block level. As files get modified, they use different blocks on their underlying device. Therefore, it is likely that the original and cloned COW images address the same blocks for different pieces of data. For similar reasons, COW disks do not help with VA creation, as multiple COW disks cannot be combined together into a single disk image.

Both the Collective [4] and Ventana [12] attempt to solve the VA maintenance problem by building upon COW concepts. Both systems enable VAs to be provisioned quickly by performing a COW copy of each VA's `system` file system. However, they suffer from the fact that they manage this file system at either the block device or monolithic file system level, providing users with only a single file system. While ideally an administrator could supply a single homogeneous shared image for all users, in practice, users want access to many heterogeneous images that must be maintained independently and therefore increase the administrator's work. The same is true for VAs provisioned by the end user, while they both enable the VAs to maintain a separate disk from the shared system disk that persists beyond upgrades.

Mirage [17] attempts to improve the disk image sprawl problem by introducing a new storage format, the Mirage Index Format (MIF), to enumerate what files belong to a package. However, it does not help with the actual image sprawl in regard to machine maintenance, because each machine reconstituted by Mirage still has a fully independent file system, as each image has its own personal copy. Although each provisioned machine can be tracked, they are now independent entities and suffer from the same problems as a traditional VA.

Stork [3] improves on package management for container-based systems by enabling containers to hard link to an underlying shared file system so that files are only stored once across all containers. By design, it cannot help with managing independent machines, virtual machines, or VAs, because hard links are a function internal to a specific file system and not usable between separate file systems.

Union file systems [11, 19] provide the ability to compose multiple different file namespaces into a single view. Unioning file systems are commonly used to provide a COW file system from a read-only copy, such as with Live-CDs. However, unioning file system by themselves do not directly help with VA management, as the underlying file system has to be maintained using regular tools. Strata builds upon and leverages this mechanism by improving its ability to handle deleted files as well as managing the layers that belong to the union. This allows Strata to provide a solution that enables efficient provisioning and management of VAs.

Strata focuses on improving virtual appliance management, but the VLFS idea can be used to address other management and security problems as well. For example, our previous work on Apiary [14] demonstrates how the VLFS can be combined with containers to provide a transparent desktop application fault containment architecture that is effective at limiting the damage from exploits to enable quick recovery while being as easy to use as a traditional desktop system.

3 Strata Basics

Figure 1 shows Strata's three architectural components: layers, layer repositories, and VLFSs. A layer is a distinct self-contained set of files that corresponds to a specific functionality. Strata classifies layers into three categories: software layers with self-contained applications and system libraries, configuration layers with configuration file changes for a specific VA, and private layers allowing each provisioned VA to be independent. Layers can be mixed and matched, and may depend on other layers. For example, a single application or system library is not fully independent, but depends on the presence of other layers, such as those that provide needed shared libraries. Strata enables layers to enumerate their dependencies on other layers. This dependency scheme

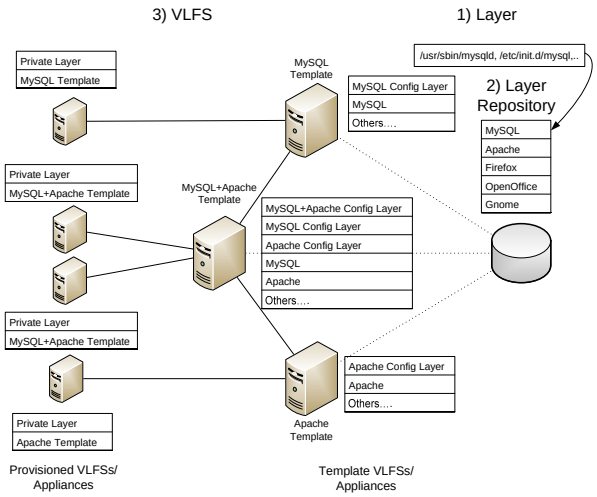

Figure 1: How Strata's Components Fit Together

allows automatic provisioning of a complete, fully consistent file system by selecting the main features desired within the file system.

Layers are provided through layer repositories. As Figure 1 shows, a layer repository is a file system share containing a set of layers made available to VAs. When an update is available, the old layer is not overwritten. Instead, a new version of the layer is created and placed within the repository, making it available to Strata's users. Administrators can also remove layers from the repository, e.g., those with known security holes, to prevent them from being used. Layer repositories are generally stored on centrally managed file systems, such as a SAN or NFS, but they can also be provided by protocols such as FTP and HTTP and mirrored locally. Layers from multiple layer repositories can form a VLFS as long as they are compatible with one another. This allows layers to be provided in a distributed manner. Layers provided by different maintainers can have the same layer names, causing a conflict. This, however, is no different from traditional package management systems as packages with the same package name, but different functionality, can be provided by different package repositories.

As Figure 1 shows, a VLFS is a collection of layers from layer repositories that are composed into a single file system namespace. The layers making up a particular VLFS are defined by the VLFS's layer definition file (LDF), which enumerates all the layers that will be composed into a single VLFS instance. To provision a VLFS, an administrator selects software layers that provide the desired functionality and lists them in the VLFS's LDF.

Within a VLFS, layers are stacked on top of another and composed into a single file system view. An implication of this composition mechanism is that layers on top can obscure files on layers below them, only allowing the contents of the file instance contained within the higher level to be used. This means that files in the private or configuration layers can obscure files in lower layers, such as when one makes a change to a default version of a configuration file located within a software layer. However, to prevent an ambiguous situation from occurring, where the file system's contents depend on the order of the software layers, Strata prevents software layers that contain a subset of the same file from being composed into a single VLFS.

4 Using Strata

Strata's usage model is centered around the usage of layers to quickly create VLFSs for VAs as shown in Figure 1. Strata allows an administrator to compose together layers to form template VAs. These template VAs can be used to form other template appliances that extend their functionality, as well as to provide the VA that end users will provision and use. Strata is designed to be used within the same setup as a traditional VM architecture. This architecture includes a cluster of physical machines that are used to host VM execution as well as a shared SAN that stores all of the VM images. However, instead of storing complete disk images on the SAN, Strata uses the SAN to store the layers that will be used by the VMs it manages.

4.1 Creating Layers and Repositories

Layers are first created and stored in layer repositories. Layer creation is similar to the creation of packages in a traditional package management system, where one builds the software, installs it into a private directory, and turns that directory into a package archive, or in Strata's case, a layer. For instance, to create a layer that contains the MySQL SQL server, the layer maintainer would download the source archive for MySQL, extract it, and build it normally. However, instead of installing it into the system's root directory, one installs it into a virtual root directory that becomes the file system component of this new layer. The layer maintainer then defines the layer's metadata, including its name (`mysql-server` in this case) and an appropriate version number to uniquely identify this layer. Finally, the entire directory structure of the layer is copied into a file system share that provides a layer repository, making the layer available to users of that repository.

4.2 Creating Appliance Templates

Given a layer repository, an administrator can then create template VAs. Creating a template VA involves: (1) Creating the template VA with an identifiable name. (2)

Determining what repositories are available to it. (3) Selecting a set of layers that provide the functionality desired.

To create a template VA that provides a MySQL SQL server, an administrator creates an appliance/VLFS named `sql-server` and selects the layers needed for a fully functional MySQL server file system, most importantly, the mysql-server layer. Strata composes these layers together into the VLFS in a read-only manner along with a read-write private layer, making the VLFS usable within a VM. The administrator boots the VM and makes the appropriate configuration changes to the template VA, storing them within the VLFS's private layer. Finally, the private layer belonging to the template appliance's VLFS is converted into the template's read-only configuration layer by being moved to a SAN file-system that the VAs can only access in a read-only manner. As another example, to create an Apache web server appliance, an administrator creates an appliance/VLFS named `web-server`, and selects the layers required for an Apache web server, most importantly, the layer containing the Apache program.

Strata extends this template model by allowing multiple template VAs to be composed together into a single new template. An administrator can create a new template VA/VLFS, `sql+web-server`, composed of the MySQL and Apache template VAs. The resulting VLFS has the combined set of software layers from both templates, both of their configuration layers, and a new configuration layer containing the configuration state that integrates the two services together, for a total of three configuration layers.

4.3 Provisioning and Running Appliance Instances

In Strata, a VLFS can be created by building off a previously defined VLFS set of layers and combining those layers with a new read-write private layer. Therefore, given previously defined templates, Strata enables VAs to be efficiently and quickly provisioned and deployed by end users. Provisioning a VA involves (1) creating a virtual machine container with a network adapter and an empty virtual disk, (2) using the network adapter's unique MAC address as the machine's identifier for identifying the VLFS created for this machine, and (3) forming the VLFS by referencing the already existing respective template VLFS and combining the template's read-only software and configuration layers with a read-write private layer provided by the VM's virtual disk.

As each VM managed by Strata does not have a physical disk off which to boot, Strata network boots each VM. When the VM boots, its BIOS discovers a network boot server which provides it with a boot image, including a base Strata environment. The VM boots this base environment, which then determines which VLFS should be mounted for the provisioned VM using the MAC address of the machine. Once the proper VLFS is mounted, the machine transitions to using it as its root file system.

4.4 Updating Appliances

Strata upgrades provisioned VAs efficiently using a simple three-step process. First, an updated layer is installed into a shared layer repository. Second, administrators are able to modify the template appliances under their control to incorporate the update. Finally, all provisioned VAs based on that template will automatically incorporate the update as well. Note that updating appliances is much simpler than updating generic machines, as appliances are not independently managed machines. This means that extra software that can conflict with an upgrade will not be installed into a centrally managed appliance. Centrally managed appliance updates are limited to changes to their configuration files and what data files they store.

Strata's updates propagate automatically even if the VA is not currently running. If a provisioned VA is shut down, the VA will compose whatever updates have been applied to its templates automatically, never leaving the file system in a vulnerable state, because it composes its file system afresh each time it boots. If it is suspended, Strata delays the update to when the VA is resumed, as updating layers is a quick task. Updating is significantly quicker than resuming, so this does not add much to its cost.

Furthermore, VAs are upgraded atomically, as Strata adds and removes all the changed layers in a single operation. In contrast, traditional package management system, when upgrading a package, first uninstalls it before reinstalling the newer version. This traditional method leaves the file system in an inconsistent state for a short period of time. For instance, when the libc package is upgraded, its contents are first removed from the file system before being replaced. Any application that tries to execute during the interim will fail to dynamically link because the main library on which it depends is not present within the file system at that moment.

4.5 Improving Security

Strata makes it much easier to manage VAs that have had their security compromised. By dividing a file system into a set of shared read-only layers and storing all file system modifications inside the private read-write layer, Strata separates changes made to the file system via layer management from regular runtime modifications. This enables Strata to easily determine when system files have

been compromised, because making a compromise persistent requires the file system be modified, modifying or adding files to the file system to create a compromise will be readily visible in the private layer. This allows Strata to not rely on tools like Tripwire [8] or maintain separate databases to determine if files have been modified from their installed state. Similarly, this check can be run external to the VA, as it just needs access to the private layer, thereby preventing an attacker from disabling it. This reduces management load due to not requiring any external databases be kept in sync with the file system state as it changes. While an attacker could try to compromise files on the shared layers, they would have to exploit the SAN containing the layer repository. In a regular virtualization architecture, if an attacker could exploit the SAN, he would also have access to all

This segregation of modified file system state also enables quick recovery from a compromised system. By sreplacing the VA's private layer with a fresh private layer, the compromised system is immediately fixed and returned to its default, freshly provisioned state. However, unlike reinstalling a system from scratch, replacing the private layer does not require throwing away the contents of the old private layer. Strata enables the layer to be mounted within the file system, enabling administrators to have easy access to the files located within it to move the uncompromised files back to their proper place.

5 Strata Architecture

Strata introduces the concept of a virtual layered file system in place of traditional monolithic file systems. Strata's VLFS allows file systems to be created by composing layers together into a single file system namespace view. Strata allows these layers to be shared by multiple VLFSs in a read-only manner or to remain read-write and private to a single VLFS.

Every VLFS is defined by a layer definition file, which specifies what software layers should be composed together. An LDF is a simple text file that lists the layers and their respective repositories. The LDF's layer list syntax is `repository/layer version` and can be proceeded by an optional modifier command. When an administrator wants to add or remove software from the file system, instead of modifying the file system directly, they modify the LDF by adding or removing the appropriate layers.

Figure 2 contains an example LDF for a MySQL SQL server template appliance. The LDF lists each individual layer included in the VLFS along with its corresponding repository. Each layer also has a number indicating which version will be composed into the file system. If an updated layer is made available, the LDF is updated

```
main/mysql-server 5.0.51a-3

main/base 1
main/libdb4.2 4.2.52-18
main/apt-utils 0.5.28.6
main/liblocale-gettext-perl 1.01-17
main/libtext-charwidth-perl 0.04-1
main/libtext-iconv-perl 1.2-3
main/libtext-wrapi18n-perl 0.06-1
main/debconf 1.4.30.13
main/tcpd 7.6-8
main/libgdbm3 1.8.3-2
main/perl 5.8.4-8
main/psmisc 21.5-1
main/libssl0.9.7 0.9.7e-3
main/liblockfile1 1.06
main/adduser 3.63
main/libreadline4 4.3-11
main/libnet-daemon-perl 0.38-1
main/libplrpc-perl 0.2017-1
main/libdbi-perl 1.46-6
main/ssmtp 2.61-2
=main/mailx 3a8.1.2-0.20040524cvs-4
```

Figure 2: LDF for MySQL Server Template

to include the new layer version instead of the old one. If the administrator of the VLFS does not want to update the layer, they can hold a layer at a specific version, with the = syntax element. This is demonstrated by the `mailx` layer in Figure 2, which is being held at the version listed in the LDF.

Strata allows an administrator to select explicitly only the few layers corresponding to the exact functionality desired within the file system. Other layers needed in the file system are implicitly selected by the layers' dependencies as described in Section 5.2. Figure 2 shows how Strata distinguishes between explicitly and implicitly selected layers. Explicitly selected layers are listed first and separated from the implicitly selected layers by a blank line. In this case, the MySQL server has only one explicit layer, mysql-server, but has 21 implicitly selected layers. These include utilities such as Perl and TCP Wrappers (tcpd), as well as libraries such as OpenSSL (libssl). Like most operating systems that require a minimal set of packages to always be installed, Strata also always includes a minimal set of shared layers that are common to all VLFSs that it denotes as `base`. In our Strata prototype, these are the layers that correspond to packages that Debian makes essential and are therefore not removable. Strata also distinguishes explicit layers from implicit layers to allow future reconfigurations to remove one implicit layer in favor of another if dependencies need to change.

When an end user provisions an appliance by cloning a template, an LDF is created for the provisioned VA. Fig-

```
@main/sql-server
```

Figure 3: LDF for Provisioned MySQL Server VA

ure 3 shows an example introducing another syntax element, `@`, that instructs Strata to reference another VLFS's LDF as the basis for this VLFS. This lets Strata clone the referenced VLFS by including its layers within the new VLFS. In this case, because the user wants only to deploy the SQL server template, this VLFS LDF only has to include the single `@` line. In general, a VLFS can reference more than one VLFS template, assuming that layer dependencies allow all the layers to coexist.

5.1 Layers

Strata's layers are composed of three components: metadata files, the layer's file system, and configuration scripts. They are stored on disk as a directory tree named by the layer's name and version. For instance, version 5.0.51a of the MySQL server, with a strata layer version of 3, would be stored under the directory `mysql-server_5.0.51a-3`. Within this directory, Strata defines a `metadata` file, a `filesystem` directory, and a `scripts` directory corresponding to the layer's three components.

The metadata files define the information that describes the layer. This includes its name, version, and dependency information. This information is important to ensure that a VLFS is composed correctly. The `metadata` file contains all the metadata that is specified for the layer. Figure 4 shows an example metadata file. Figure 5 shows the full metadata syntax. The metadata file has a single field per line with two elements, the field type and the field contents. In general, the metadata file's syntax is `Field Type: value`, where value can be either a single entry or a comma-separated list of values.

The layer's file system is a self-contained set of files providing a specific functionality. The files are the individual items in the layer that are composed into a larger VLFS. There are no restrictions on the types of files that can be included. They can be regular files, symbolic links, hard links, or device nodes. Similarly, each directory entry can be given whatever permissions are appropriate. A layer can be seen as a directory stored on the shared file system that contains the same file and directory structure that would be created if the individual items were installed into a traditional file system. On a traditional UNIX system, the directory structure would typically contain directories such as `/usr`, `/bin` and `/etc`. Symbolic links work as expected between layers since they work on path names, but one limitation is that hard links cannot exist between layers.

The layer's configuration scripts are run when a layer is added or removed from a VLFS to allow proper integration of the layer within the VLFS. Although 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 should register itself with the system's MIME database to allow programs contained within the layer to be launched automatically when a user wants to play an `mp3` file. Similarly, if the layer were removed, it should remove the programs contained within itself from the MIME database.

```
Layer: mysql-server
Version: 5.0.51a-3
Depends: ..., perl (>= 5.6),
  tcpd (>= 7.6-4),...
```

Figure 4: Metadata for MySQL-Server Layer

```
Layer: Layer Name
Version: Version of Layer Unit
Conflicts: layer1 (opt. constraint), ...
Depends: layer1 (...),
   layer2 (...) | layer3, ...
Pre-Depends: layer1 (...), ...
Provides: virtual_layer, ...
```

Figure 5: Metadata Specification

Strata supports four types of configuration scripts: pre-remove, post-remove, pre-install, and post-install. If they exist in a layer, the appropriate script is run before or after a layer is added or removed. For example, a pre-remove script can be used to shut down a daemon before it is actually removed, while a post-remove script can be used to clean up file system modifications in the private layer. Similarly, a pre-install script can ensure that the file system is as the layer expects, while the post-install script can start daemons included in the layer. The configuration scripts can be written in any scripting language. The layer must include the proper dependencies to ensure that the scripting infrastructure is composed into the file system in order to allow the scripts to run.

5.2 Dependencies

A key Strata metadata element is enumeration of the dependencies that exist between layers. Strata's dependency scheme is heavily influenced by the dependency scheme in Linux distributions such as Debian and Red Hat. In Strata, every layer composed into Strata's VLFS is termed a *layer unit*. Every layer unit is defined by its name and version. Two layer units that have the same name but different layer versions are different units of the same layer. A *layer* refers to the set of layer units of a particular name. Every layer unit in Strata has a set of dependency constraints placed within its metadata. There are four types of dependency constraints: (a) de-

pendency, (b) pre-dependency, (c) conflict and (d) provide.

Dependency and Pre-Dependency: Dependency and pre-dependency constraints are similar in that they require another layer unit to be integrated at the same time as the layer unit that specifies them. They differ only in the order the layer's configuration scripts are executed to integrate them into the VLFS. A regular dependency does not dictate order of integration. A pre-dependency dictates that the dependency has to be integrated before the dependent layer. Figure 4 shows that the MySQL layer depends on TCP Wrappers, (`tcpd`), because it dynamically links against the shared library `libwrap.so.0` provided by TCP Wrappers. MySQL cannot run without this shared library, so the layer units that contain MySQL must depend on a layer unit containing an appropriate version of the shared library. These constraints can also be versioned to further restrict which layer units satisfy the constraint. For example, shared libraries can add functionality that breaks their application binary interface (ABI), breaking in turn any applications that depend on that ABI. Since MySQL is compiled against version 0.7.6 of the libwrap library, the dependency constraint is versioned to ensure that a compatible version of the library is integrated at the same time.

Conflict: Conflict constraints indicate that layer units cannot be integrated into the same VLFS. There are multiple reasons this can occur, but it is generally because they depend on exclusive access to the same operating system resource. This can be a TCP port in the case of an Internet daemon, or two layer units that contain the same file pathnames and therefore would obscure each other. For this reason, Strata defines that two layer units of the same layer are by definition in conflict because they will contain some of the same files.

An example of this constraint occurs when the ABI of a shared library changes without any source code changes, generally due to an ABI change in the tool chain that builds the shared library. Because the ABI has changed, the new version can no longer satisfy any of the previous dependencies. But because nothing else has changed, the file on disk will usually not be renamed either. A new layer must then be created with a different name, ensuring that the library with the new ABI is never used to satisfy an old dependency on the original layer. Because the new layer contains the same files as the old layer, it must conflict with the older layer to ensure that they are not integrated into the same file system.

Provide: Provide dependency constraints introduce virtual layers. A regular layer provides a specific set of files, but a virtual layer indicates that a layer provides a particular piece of general functionality. Layer units that depend on a certain piece of general functionality can depend on a specific virtual layer name in the normal manner, while layer units that provide that functionality will explicitly specify that they do. For example, layer units that provide HTML documentation depend on the presence of a web server to enable a user to view them, but which one is not important. Instead of depending on a particular web server, they depend on the virtual layer name `httpd`. Similarly, layer units containing a web server and obeying system policy for the location of static html content, such as Apache or Boa, are defined to provide the `httpd` virtual layer name and therefore satisfy those dependencies. Unlike regular layer units, virtual layers are not versioned.

Example: Figure 2 shows how dependencies can affect a VLFS in practice. This VLFS has only one explicit layer, mysql-server, but 21 implicitly selected layers. The mysql-server layer itself has a number of direct dependencies, including Perl, TCP Wrappers, and the mailx program. These dependencies in turn depend on the Berkeley DB library and the GNU dbm library, among others. Using its dependency mechanism, Strata is able to automatically resolve all the other layers needed to create a complete file system by specifying just a single layer

Returning to Figure 4, this example defines a subset of the layers that the mysql-server layer requires to be composed into the same VLFS to allow MySQL to run correctly. More generally, Figure 5 shows the complete syntax for the dependency metadata. Provides is the simplest, with only a comma separated list of virtual layer names. Conflicts adds an optional version constraint to each conflicted layer to limit the layer units that are actually in conflict. Depends and Pre-Depends add a boolean OR of multiple layers in their dependency constraints to allow multiple layers to satisfy the dependency.

Resolving Dependencies: To allow an administrator to select only the layers explicitly desired within the VLFS, Strata automatically resolves dependencies to determine which other layers must be included implicitly.

Linux distributions already face this problem and tools have been developed to address it, such as Apt [2] and Smart [10]. To leverage Smart, Strata adopts the same metadata database format that Debian uses for packages for its own layers. In Strata, when an administrator requests that a layer be added to or removed from a template appliance, Smart also evaluates if the operation can succeed and what is the best set of layers to add or remove. Instead of acting directly on the contents of the file system, however, Strata only has to update the template's VLFS's definition file with the set of layers to be composed into the file system.

5.3 Layer Creation

Strata allows layers to be created in two ways. First, Strata allows the `.deb` packages used by Debian-derived distributions and the `.rpm` packages used by RedHat-derived distributions to be converted into layers that Strata users can use. Strata converts packages into layers in two steps. First, Strata extracts the relevant metadata from the package, including its name and version. Second, Strata extracts the package's file contents into a private directory that will be the layer's file system components. When using converted packages, Strata leverages the underlying distribution's tools to run the configuration scripts belonging to the newly created layers correctly. Instead of using the distribution's tools to unpack the software package, Strata composes the layers together and uses the distribution's tools as though the packages have already been unpacked. Although Strata is able to convert packages from different Linux distributions, it cannot mix and match them because they are generally ABI incompatible with one another.

More commonly, Strata leverages existing packaging methodologies to simplify the creation of layers from scratch. In a traditional system, when administrators install a set of files, they copy the files into the correct places in the file system using the root of the file system tree as their starting point. For instance, an administrator might run `make install` to install a piece of software compiled on the local machine. But in Strata layer creation is a three step process. First, instead of copying the files into the root of the local file system, the layer creator installs the files into their own specific directory tree. That is, they make a blank directory to hold a new file system tree that is created by having the `make install` copy the files into a tree rooted at that directory, instead of the actual file system root.

Second, the layer maintainer extracts programs that integrate the files into the underlying file system and creates scripts that run when the layer is added to and removed from the file system. Examples of this include integration with Gnome's GConf configuration system, creation of encryption keys, or creation of new local users and groups for new services that are added. This leverages skills that package maintainers in a traditional package management world already have.

Finally, the layer maintainer needs to set up the metadata correctly. Some elements of the metadata, such as the name of the layer and its version, are simple to set, but dependency information can be much harder. But because package management tools have already had to address this issue, Strata is able to leverage the tools they have built. For example, package management systems have created tools that infer dependencies using an executable dynamically linking against shared libraries [15]. Instead of requiring the layer maintainer to enumerate each shared library dependency, we can programmatically determine which shared libraries are required and populate the dependency fields based on those versions of the library currently installed on the system where the layer is being created.

5.4 Layer Repositories

Strata provides local and remote layer repositories. Local layer repositories are provided by locally accessible file system shares made available by a SAN. They contain layer units to be composed into the VLFS. This is similar to a regular virtualization infrastructure in which all the virtual machines' disks are stored on a shared SAN. Each layer unit is stored as its own directory; a local layer repository contains a set of directories, each of which corresponds to a layer unit. The local layer repository's contents are enumerated in a database file providing a flat representation of the metadata of all the layer units present in the repository. The database file is used for making a list of what layers can be installed and their dependency information. By storing the shared layer repository on the SAN, Strata lets layers be shared securely among different users' appliances. Even if the machine hosting the VLFS is compromised, the read-only layers will stay secure, as the SAN will enforce the read-only semantic independently of the VLFS.

Remote layer repositories are similar to local layer repositories, but are not accessible as file system shares. Instead, they are provided over the Internet, by protocols such as FTP and HTTP, and can be mirrored into a local layer repository. Instead of mirroring the entire remote repository, Strata allows on-demand mirroring, where all the layers provided by the remote repository are accessible to the VAs, but must be mirrored to the local mirror before they can be composed into a VLFS. This allows administrators to store only the needed layers while maintaining access to all the layers and updates that the repository provides. Administrators can also filter which layers should be available to prevent end users from using layers that violate administration policy. In general, an administrator will use these remote layer repositories to provide the majority of layers, much as administrators use a publicly managed package repository from a regular Linux distribution.

Layer repositories let Strata operate within an enterprise environment by handling three distinct yet related issues. First, Strata has to ensure that not all end users have access to every layer available within the enterprise. For instance, administrators may want to restrict certain layers to certain end users for licensing or security reasons. Second, as enterprises get larger, they gain levels of administration. Strata must support the creation of an

enterprise-wide policy while also enabling small groups within the enterprise to provide more localized administration. Third, larger enterprises supporting multiple operating systems cannot rely on a single repository of layers because of inherent incompatibilities among operating systems.

By allowing a VLFS to use multiple repositories, Strata solves these three problems. First, multiple repositories let administrators compartmentalize layers according to the needs of their end users. By providing end users with access only to needed repositories, organizations prevent their end users from using the other layers. Strata depends on traditional file system access control mechanisms to enforce these permissions. Second, by allowing sub-organizations to set up their own repositories, Strata lets a sub-organization's administrator provide the layers that end users need without requiring intervention by administrators of global repositories. Finally, multiple repositories allow Strata to support multiple operating systems, as each distinct operating system has its own set of layer repositories.

5.5 VLFS Composition

To create a VLFS, Strata has to solve a number of file system-related problems. First, Strata has to support the ability to combine numerous distinct file system layers into a single static view. This is equivalent to installing software into a shared read-only file system. Second, because users expect to treat the VLFS as a normal file system, for instance, by creating and modifying files, Strata has to let VLFSs be fully modifiable. By the same token, users must also be able to delete files that exist on the read-only layer.

By basing the VLFS on top of unioning file systems [11, 19], Strata solves all these problems. Unioning file systems join multiple layers into a single namespace. Unioning file systems have been extended to apply attributes such as read-only and read-write to their layers. The VLFS leverages this property to force shared layers to be read-only, while the private layer remains read-write. If a file from a shared read-only layer is modified, it is copied-on-write (COW) to the private read-write layer before it is modified. For example, Live-CDs use this functionality to provide a modifiable file system on top of the read-only file system provided by the CD. Finally, unioning file systems use white-outs to obscure files located on lower layers. For example, if a file located on a read-only layer is deleted, a white-out file will be created on the private read-write layer. This file is interpreted specially by the file-system and is not revealed to the user while also preventing the user from seeing files with the same name.

But end users need to be able to recover deleted files by reinstalling or upgrading the layer containing them. This is equivalent to deleting a file from a traditional monolithic file system, but reinstalling the package containing the file in order to recover it. Also, Strata supports adding and removing layers dynamically without taking the file system off line. This is equivalent to installing, removing, or upgrading a software package while a monolithic file system is online.

Unlike a traditional file system, where deleted system files can be recovered simply by reinstalling the package containing that file, in Strata, white-outs in the private layer persist and continue to obscure the file even if the layer is replaced. To solve this problem, Strata provides a VLFS with additional writeable layers associated with each read-only shared layer. Instead of containing file data, as does the topmost private writeable layer, these layers just contain white-out marks that will obscure files contained within their associated read-only layer. The user can delete a file located in a shared read-only layer, but the deletion only persists for the lifetime of that particular instance of the layer. When a layer is replaced during an upgrade or reinstall, a new empty white-out layer will be associated with the replacement, thereby removing any preexisting white-outs. In a similar way, Strata handles he case where a file belonging to a shared read-only layer is modified and therefore copied to the VLFS's private read-write layer. Strata provides a *revert* command that lets the owner of a file that has been modified revert the file to its original pristine state. While a regular VLFS *unlink* operation would have removed the modified file from the private layer and created a white-out mark to obscure the original file, *revert* only removes the copy in the private layer, thereby revealing the original below it.

Strata also allows a VLFS to be managed while being used. Some upgrades, specifically of the kernel, will require the VA to be rebooted, but most should be able to occur without taking the VA off line. However, if a layer is removed from a union, the data is effectively removed as well because unions operate only on file system namespaces and not on the data the underlying files contain. If an administrator wants to remove a layer from the VLFS, they must take the VA off line, because layers cannot be removed while in use.

To solve this problem, Strata emulates a traditional monolithic file system. When an administrator deletes a package containing files in use, the processes that are currently using those files will continue to work. This occurs by virtue of `unlink`'s semantic of first removing a file from the file system's namespace, and only removing its data after the file is no longer in use. This lets processes continue to run because the files they need will not be removed until after the process terminates. This creates a semantic in which a currently running program

can be using versions of files no longer available to other programs.

Existing package managers use this semantic to allow a system to be upgraded online, and it is widely understood. Strata applies the same semantic to layers. When a layer is removed from a VLFS, Strata marks the layer as `unlinked`, removing it from the file system namespace. Although this layer is no longer part of the file system namespace and thus cannot be used by any operations such as `open` that work on the namespace, it does remain part of the VLFS, enabling data operations such as `read` and `write` to continue working correctly for previously opened files.

6 Experimental Results

We have implemented Strata's VLFS as a loadable kernel module on an unmodified Linux 2.6 series kernel as well as a set of userspace management tools. The file system is a stackable file system and is an extended version of UnionFS [19]. We present experimental results using our Strata Linux prototype to manage various VAs, demonstrating its ability to reduce management costs while incurring only modest performance overhead. Experiments were conducted on VMware ESX 3.0 running on an IBM BladeCenter with 14 IBM HS20 eServer blades with dual 3.06 GHz Intel Xeon CPUs, 2.5 GB RAM, and a Q-Logic Fibre Channel 2312 host bus adapter connected to an IBM ESS Shark SAN with 1 TB of disk space. The blades were connected by a gigabit Ethernet switch. This is a typical virtualization infrastructure in an enterprise computing environment where all virtual machines are centrally stored and run. We compare plain Linux VMs with a virtual block device stored on the SAN and formatted with the ext3 file system to VMs managed by Strata with the layer repository also stored on the SAN. By storing both the plain VM's virtual block device and Strata's layers on the SAN, we eliminate any differences in performance due to hardware architecture.

To measure management costs, we quantify the time taken by two common tasks, provisioning and updating VAs. We quantify the storage and time costs for provisioning many VAs and the performance overhead for running various benchmarks using the VAs. We ran experiments on five VAs: an Apache web server, a MySQL SQL server, a Samba file server, an SSH server providing remote access, and a remote desktop server providing a complete GNOME desktop environment. While the server VAs had relatively few layers, the desktop VA has very many layers. This enables the experiments to show how the VLFS performance scales as the number of layers increases. To provide a basis for comparison, we provisioned these VAs using (1) the normal VMware virtualization infrastructure and plain Debian package management tools, and (2) Strata. To make a conservative comparison to plain VAs and to test larger numbers of plain VAs in parallel, we minimized the disk usage of the VAs. The desktop VA used a 2 GB virtual disk, while all others used a 1 GB virtual disk.

	Apache	**MySQL**	**Samba**	**SSH**	**Desktop**
Plain	184s	179s	183s	174s	355s
Strata	0.002s	0.002s	0.002s	0.002s	0.002s
QCOW2	0.003s	0.003s	0.003s	0.003s	0.003s

Table 1: VA Provisioning Times

6.1 Reducing Provisioning Times

Table 1 shows how long it takes Strata to provision VAs versus regular and COW copying. To provision a VA using Strata, Strata copies a default VMware VM with an empty sparse virtual disk and provides it with a unique MAC address. It then creates a symbolic link on the shared file system from a file named by the MAC address to the layer definition file that defines the configuration of the VA. When the VA boots, it accesses the file denoted by its MAC address, mounts the VLFS with the appropriate layers, and continues execution from within it. To provision a plain VA using regular methods, we use QEMU's `qemu-img` tool to create both raw copies and COW copies in the QCOW2 disk image format.

Our measurements for all five VAs show that using COW copies and Strata takes about the same amount of time to provision VAs, while creating a raw image takes much longer. Creating a raw image for a VAs takes 3 to almost 6 minutes and is dominated by the cost of copying data to create a new instance of the VA. For larger VAs, these provisioning times would only get worse. In contrast, Strata provisions VAs in only a few milliseconds because a null VMware VM has essentially no data to copy. Layers do not need to be copied, so copying overhead is essentially zero. While COW images can be created in a similar amount of time, they do not provide any of the management benefits of Strata, as each new COW image is independent of the base image from which it was created.

6.2 Reducing Update Times

Table 2 shows how long it takes to update VAs using Strata versus traditional package management. We provisioned ten VA instances each of Apache, MySQL, Samba, SSH, and Desktop for a total of 50 provisioned VAs. All were kept in a suspended state. When a security patch was made available for the `tar` package installed in all the VAs, we updated them [18]. Strata simply updates the layer definition files of the VM templates, which it can do even when the VAs are not active. When the VA is later resumed during normal operation,

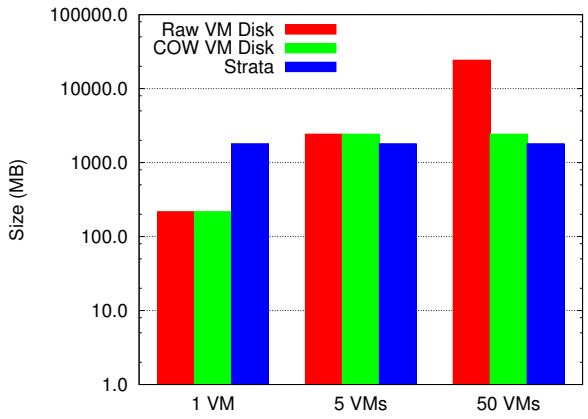

Figure 6: Storage Requirements

it automatically checks to see if the layer definition file has been updated and updates the VLFS namespace view accordingly, an operation that is measured in microseconds. To update a plain VA using normal package management tools, each VA instance needs to be resumed and put on the network. An administrator or script must ssh into each VA, fetch and install the update packages from a local Debian mirror, and finally re-suspend the VA.

Table 2 shows the total average time to update each VA using traditional methods versus Strata. We break down the update time into times to resume the VM, get access to the network, actually perform the update, and re-suspend the VA. The measurements show that the cost of performing an update is dominated by the management overhead of preparing the VAs to be updated and not the update itself. Preparation is itself dominated by getting an IP address and becoming accessible on a busy network. While this cost is not excessive on a quiet network, on a busy network it can take a significant amount of time for the client to get a DHCP address, and for the ARP on the machine controlling the update to find the target machine. The average total time to update each plain VA is about 73 seconds. In contrast, Strata takes only a second to update each VA. As this is an order of magnitude shorter even than resuming the VA, Strata is able to delay the update to a point when the VA will be resumed from standby normally without impacting its ability to quickly respond. Strata provides over 70 times faster update times than traditional package management when managing even a modest number of VAs. Strata's ability to decrease update times would only improve as the number of VAs being managed grows.

	Plain	Strata
VM Wake	14.66s	NA
Network	43.72s	NA
Update	10.22s	1.041s
Suspend	3.96s	NA
Total	73.2s	1.041s

Table 2: VA Update Times

6.3 Reducing Storage Costs

Figure 6 shows the total storage space required for different numbers of VAs stored with raw and COW disk images versus Strata. We show the total storage space for 1 Apache VA, 5 VAs corresponding to an Apache, MySQL, Samba, SSH, and Desktop VA, and 50 VAs corresponding to 10 instances of each of the 5 VAs. As expected, for raw images, the total storage space required grows linearly with the number of VA instances. In contrast, the total storage space using COW disk images and Strata is relatively constant and independent of the number of VA instances. For one VA, the storage space required for the disk image is less than the storage space required for Strata, as the layer repository used contains more layers than those used by any one of the VAs. In fact, to run a single VA, the layer repository size could be trimmed down to the same size as the traditional VA.

For larger numbers of VAs, however, Strata provides a substantial reduction in the storage space required, because many VAs share layers and do not require duplicate storage. For 50 VAs, Strata reduces the storage space required by an order of magnitude over the raw disk images. Table 3 shows that there is much duplication among statically provisioned virtual machines, as the layer repository of 405 distinct layers needed to build the different VLFSs for multiple services is basically the same size as the largest service. Although initially Strata does not have an significant storage benefit over COW disk images, as each COW disk image is independent from the version it was created from, it now must be managed independently. This increases storage usage, as the same updates must be independently applied to many independent disk images

6.4 Virtualization Overhead

To measure the virtualization cost of Strata's VLFS, we used a range of micro-benchmarks and real application workloads to measure the performance of our Linux Strata prototype, then compared the results against vanilla Linux systems within a virtual machine. The virtual machine's local file system was formatted with the `Ext3` file system and given read-only access to a SAN partition formatted with `Ext3` as well. We performed each benchmark in each scenario 5 times and provide the average of the results.

Repo	**Apache**	**MySQL**	**Samba**	**SSH**	**Desktop**
1.8GB	217MB	206MB	169MB	127MB	1.7GB
# Layer	43	23	30	12	404
Shared	191MB	162MB	152MB	123MB	169MB
Unique	26MB	44MB	17MB	4MB	1.6GB

Table 3: Layer Repository vs. Static VAs

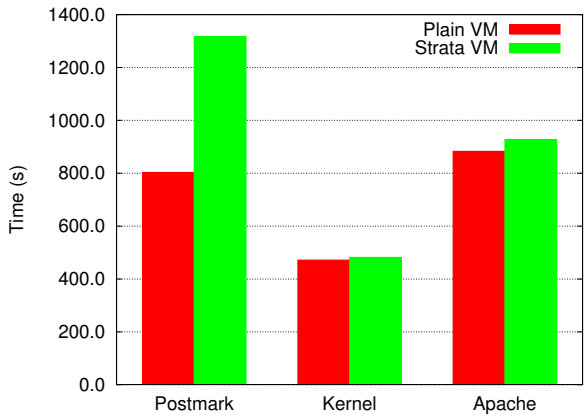

Figure 7: Application Benchmarks

To demonstrate the effect that Strata's VLFS has on system performance, we performed a number of benchmarks. Postmark [7], the first benchmark, 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 files, and performed 200,000 transactions. Postmark is very intensive on a few specific file system operations such as `lookup()`, `create()`, and `unlink()`, because it is constantly creating, opening, and removing files. Figure 7 shows that running this benchmark within a traditional VA is significantly faster than running it in Strata. This is because as Strata composes multiple file system namespaces together, it places significant overhead on those namespace operations.

To demonstrate that postmark's results are not indicative of application oriented performance, we ran two application benchmarks to measure the overhead Strata imposes in a desktop and server VA scenario. The first benchmark was a multi-threaded build of the Linux 2.6.18.6 kernel with two concurrent jobs using the two CPUs allocated to the VM. In all scenarios, we added the 8 software layers required to build a kernel to the layers needed to provide the service. Figure 7 shows that while Strata imposes a slight overhead on the kernel build compared to the underlying file system it uses, the cost is minimal, under 5% at worst.

The second benchmark measured the amount of HTTP transactions that were able to be completed per second to an Apache web server placed under load. We imported the database of a popular guitar tab search engine and used the `http_load` [13] benchmark to continuously performed a set of 20 search queries on the database until 60,000 queries in total have been performed. For each case that did not already contain Apache, we added the appropriate layers to the layer definition file to make Apache available. Figure 7 shows that Strata imposes a minimal overhead of only 5%.

While the Postmark benchmark demonstrated that the VLFS is not an appropriate file system for workloads that are heavy with namespace operations, this shouldn't prevent Strata from being used in those scenarios. No file system is appropriate for all workloads and no system has to be restricted to simply using one file system. One can use Strata and the VLFS to manage the system's configuration while also providing an additional traditional file system on a seperate partition or virtual disk drive to avoid all the overhead the VLFS imposes. This will be very effective for workloads, such as the mail server Postmark is emulating, where namespace heavy operations, such as a mail server processing its mail queue, can be kept on a dedicated file system.

7 Conclusions and Future Work

Strata introduces a new and better way for system administrators to manage virtual appliances using virtual layered file systems. Strata integrates package management semantics with the file system by using a novel form of file system unioning enable dynamic composition of file system layers. This provides powerful new management functionality for provisioning, upgrading, securing, and composing VAs. VAs can be quickly and simply provisioned as no data needs to be copied into place. VAs can be easily upgraded as upgrades can be done once centrally and applied atomically, even for a heterogeneous mix of VAs and when VAs are suspended or turned off. VAs can be more effectively secured since file system modifications are isolated so compromises can be easily identified. VAs can be composed as building blocks to create new systems since file system composition also serves as the core mechanism for creating and maintaining VAs. We have implemented Strata on Linux by providing the VLFS as a loadable kernel modules, but without requiring any source code level kernel changes, and have demonstrated how a Strata can be used in real life situations to improve the ability of system administrators to manage systems. Strata significantly reduces the amount of disk space required for multiple VAs, and allows them to be provisioned almost instantaneously and quickly updated no matter how many are in use.

While Strata just exists as a lab prototype today, there are few steps that could make it significantly more deployable. First, our changes to UnionFS should either be integrated with the current version of UnionFS or with another unioning file system. Second, better tools should be created for managing the creation and management of individual layers. This can include better tools for converting layers from existing Linux distributions as well as new tools that enable layers to be created in a way that takes full advantage of Strata's concepts. Third, the ability to to integrate Strata's concepts with cloud com-

puting infrastructures, such as Eucalyptus, should be investigated.

Acknowledgments

Carolyn Rowland provided helpful comments on earlier drafts of this paper. This work was supported in part by AFOSR MURI grant FA9550-07-1-0527 and NSF grants CNS-1018355, CNS-0914845, and CNS-0905246.

References

[1] The RPM Package Manager. `http://www.rpm.org/`.

[2] B. Byfield. An Apt-Get Primer. `http://www.linux.com/articles/40745`, Dec. 2004.

[3] J. Capps, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. H. Hartman. Stork: Package Management for Distributed VM Environments. In *The 21st Large Installation System Administration Conference*, Dallas, TX, Nov. 2007.

[4] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *The 2nd Symposium on Networked Systems Design and Implementation*, pages 259–272, Boston, MA, Apr. 2005.

[5] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[6] J. Fernandez-Sanguino. Debian GNU/Linux FAQ - Chapter 8 - The Debian Package Management Tools. `http://www.debian.org/doc/FAQ/ch-pkgtools.en.html`.

[7] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR3022, Network Appliance, Inc., Oct. 1997.

[8] G. Kim and E. Spafford. Experience with Tripwire: Using Integrity Checkers for Intrusion Detection. In *The 1994 System Administration, Networking, and Security Conference*, Washington, DC, Apr. 1994.

[9] M. McLoughlin. QCOW2 Image Format. `http://www.gnome.org/~markmc/qcow-image-format.htm`, Sept. 2008.

[10] G. Niemeyer. Smart Package Manager. `http://labix.org/smart`.

[11] J.-S. Pendry and M. K. McKusick. Union Mounts in 4.4BSD-lite. In *The 1995 USENIX Technical Conference*, New Orleans, LA, Jan. 1995.

[12] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *3rd Symposium on Networked Systems Design and Implementation*, pages 353–366, San Jose, CA, May 2006.

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

[14] S. Potter and J. Nieh. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *The 2010 USENIX Annual Technical Conference*, pages 103–116, June 2010.

[15] D. Project. DDP Developers' Manuals. `http://www.debian.org/doc/devel-manuals`.

[16] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *1st USENIX conference on File and Storage Technologies*, Monterey, CA, Jan. 2002.

[17] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl. In *The 2008 ACM International Conference on Virtual Execution Environments*, pages 111–120, Seattle, WA, Mar. 2008.

[18] F. Weimer. DSA-1438-1 Tar – Several Vulnerabilities. `http://www.ua.debian.org/security/2007/dsa-1438`, Dec. 2007.

[19] 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*, 2(1):1–32, Feb. 2006.