Improving Virtual Appliance Management through Virtual Layered File Systems

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

Columbia Univeristy Technical Report CUCS-008-09, January 2009

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 present Strata, a system that introduces the Virtual Layered File System (VLFS) and integrates it with virtual appliances to simplify system management. Unlike a traditional file system, which is a monolithic entity, a VLFS is a collection of individual software layers composed together to provide the traditional file system view. Individual layers are maintained in a central repository and shared across all VLFSs that use them. Layer changes and upgrades only need to be done once in the repository and are then automatically propagated to all VLFSs, resulting in management overhead independent of the number of virtual appliances. We have implemented a Strata Linux prototype without any application or operating system kernel changes. 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

The rise of virtualization has resulted in the growing adoption and use of virtual appliances (VAs). VAs are pre-built software bundles run inside virtual machines (VMs). For example, one VM might be tailored to be a web server VA, another might be tailored to be a desktop computing VA. 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. Because of its greater simplicity, an individual VA may often be easier to manage than a general-purpose machine.

VAs simplify application deployment. Once an application is installed in a VA, it can be easily copied and deployed by end users with minimal hassle since both the software and its configuration have already been setup in the VA. Furthermore, a new VA can be easily created by just cloning an existing VA that already contains a base installation of the necessary software, then modifying it by adding or removing applications and changing the system configuration. There is no need to set up the common components again from scratch.

The utility of VAs and the ease with which they can be created and deployed is fueling their rapid proliferation in the enterprise. As VAs are cloned and modified, creating an ever increasing sprawl of different configurations, organizations which once had a few hardware machines to manage now find themselves juggling many more VAs with diverse system configurations and software installations. VAs are increasingly networked, which only complicates the management problem, given the myriad of viruses and other attacks commonplace today. Security problems can wreak havoc on an organization's virtual computing infrastructure. While software patches are released for these threats, the need to constantly deploy patches and upgrade software creates a management nightmare as the number of VAs in the enterprise continues to rise. Many VAs may be turned off, suspended, or not even actively managed, making patch deployment before a security problem hits even more difficult. Although the management of any one VA may not be difficult, the need to manage many different VAs results in a huge scaling problem for large organizations as management overhead grows linearly with the number of VAs that need to be maintained.

Many approaches have tried to address these problems, including traditional package management systems [7, 16], using copy-on-write disks [10], and new VM storage formats [13, 3]. Unfortunately, these approaches suffer from various drawbacks that limit their utility and effectiveness in practice. They either incur management overheads that grow with the number of VAs, or require all VAs to have the same configuration, eliminating the key benefits of VAs. The fundamental problem with previous approaches is that they are based on the concept of a monolithic file system or block device. These file systems and block devices address their data at the block layer. 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 need the same maintenance.

To address this management problem, we present Strata, a system that decomposes monolithic file systems into collections of files that can be shared across different VAs and centrally maintained to simplify system management. These collection of files are dynamically composed back together in each VA to provide the traditional file system view applications expect. This is done by providing a file system that addresses its contents by file location.

Strata is built around three architectural components: layers, layer repositories, and virtual layered file systems. A layer is a file hierarchy of related files that are typically changed or upgraded as a unit, such as a software package or application. A layer may require other layers to function correctly in the same manner that applications often require various system libraries to run. Strata associates dependency information with each layer that defines relationships among distinct layers.

A layer repository is used to centrally store layers. Layers are updated and maintained in the layer repository. For example, if a new version of an application becomes available, a new layer is added to the repository. If a patch for an application is issued, the corresponding layer is patched by creating a new layer with the patch. 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.

A VLFS is the file system for a VA. Unlike a traditional file system that is a monolithic entity, it is a collection of individual layers dynamically composed together into a single file system view. Each VA has its own VLFS, which typically consists of a private read-write layer and a set of read-only layers shared through the layer repository. The private read-write layer is used for all file system modifications private to the VA, such as modifying user data. The shared read-only layers allow VAs with very different system configurations and applications to still share common layers that represent common software components used across VAs. Layer changes to shared layers only need to be done once in the repository and are then automatically propagated to all VLFSs, resulting in management overhead independent of the number of VAs.

By dynamically building a VLFS out of discrete layers, Strata introduces file system composition as the package management semantic to provide a number of management benefits. First, Strata enables faster and easier ways to provision VAs. All an administrator needs to do to provision a VA is select the applications and tools of interest from the layer repository. The VA's VLFS automatically composes the selected layers together with a read-write layer and incorporates any additional layers needed to resolve any necessary dependencies. The VA is then immediately ready to use. Since layers are stored in the shared layer repository, provisioning VAs is very fast as no data has to be copied into place. As the layer repository allows easy identification of the applications and tools of interest, and the VLFS resolves dependencies on other layers automatically, provisioning VAs is relatively easy. Since VAs are simply defined by the sets of layers associated with them,, Strata also enables a new way to build VAs by compositing existing VAs together.

Second, Strata simplifies upgrades and maintenance of VAs. If a layer contains a bug that has to be fixed, all an administrator has to do is create a replacement layer that contains the bug fix and inform the provisioned VA to incorporate the layer into the VLFS's namespace view. Unlike traditional VAs that have to reboot themselves when an upgrade occurs, Strata enables online upgrades. New layers are dynamically incorporated into a VLFS on the fly, much like a traditionally managed machine incorporates updated packages.

Finally, Strata enables easier recovery of VAs in the presence of security exploits. A VLFS enables systems to distinguish between the deployed system's initial state and changes that are made to it over time as modifications to the file system from the VA are performed in its private read-write layer. If a VA is compromised and an attacker attempts to install new malware or modify existing applications, these changes will be separated from the deployed system's initial state and isolated to the read-write layer. Such changes can then be more easily identified and removed to return the VA to a clean state.

We have implemented a Strata Linux prototype without any application or operating system kernel changes. We show that layers can be implemented by simply combining traditional package management with file system unioning in a novel way to provide powerful new functionality. 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. On Linux, this would generally be Debian's Package Manager (dpkg) [7] or the Red Hat Package Manager (rpm) [16]. Package managers perform a number of operations when packages are installed or removed from the file system: (1) they determine if a specific package can be installed into or removed from the file system, (2) they unpack a package's contents into the file system on installation and delete them on removal, and (3) they use configuration scripts to configure the system its being installed into or removed from appropriately.

Package managers suffer from a number of flaws for managing VAs. They are not space or time efficient, as each provisioned VA needs an independent copy of the package's files and 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. Provisioning takes a significant amount of time as it involves downloading, extracting and installing a large number of packages, as well as requiring a significant amount of space. Even for existing VAs, if an update has to be applied to all of them, such as to fix a critical security hole in a common system library, a significant time is spent in aggregate installing the update into every individual VA. Finally, package management systems work in the context of a running system to modify the file system directly. These standard tools often do not work outside the context of a running system, for example for a VA that 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 [10] format. A VA can then be provisioned quickly as very 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 and is equivalent to a regular copy and therefore suffers from all the same maintenance problem 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 will use different blocks on their underlying device. Therefore, its very likely that the original and cloned COW image will be addressing 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 [3] and Ventana [13] attempt to solve the VA maintenance problem by building upon COW concepts. The Collective uses shared virtual disks to improve the ability of system administrators to manage large number of virtual machine instances, while Ventana works at the file system layer. Both systems work by providing virtual machines with two file systems, a `system` file system that is shared in a COW manner and a private `user` file system to contain data that persists past an upgrade. Both systems enable VAs to be provisioned quickly by just 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 just 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 need to be maintained independently and therefore increase the work of the administrator. Similarly, for end user provisioned VAs, while they both enable the VAs to maintain a separate disk from the shared system disk that will persist beyond upgrades, they both suffer the same problems as traditional VAs due to the monolithic nature of the system disk. Any changes to the system disk will be lost on upgrade and it cannot be upgraded online due to it being fully replaced.

Mirage [15] 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. Instead of having a single package that contains the files that belong to it, Mirage has a single store that contains all files belonging to all packages. The files that belong to a package are then indexed into the shared storage medium by a manifest file that determines which files in the shared store belong to the package. Users create a virtual disk by specifying which packages should be installed, and the manifest files determines which files should be installed into the file system, and those files are copied into place. The Mirage format provides significant benefits in certain specific areas. It improves the efficiency of the back end storage, as many files that would otherwise be duplicated are no longer duplicated. Similarly, it enables an administrator to easily determine what programs are in a disk images. However, it does not help with the actual image sprawl in regards to machine maintenance, as each machine reconstituted by Mirage still has a fully independent file system, as each image has its own personal copy. While each provisioned machine can be tracked, they are now independent entities and suffer from the same problems as a traditional VAs.

Stork [2] 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.

Union file systems [12, 19] provide the ability to compose multiple different file system namespaces into a single namespace view. While file system unioning does not help directly with VA management, Strata builds upon and leverages this mechanism to provide a solution that enables efficient provisioning and management of VAs.

3 Strata Basics

Figure 1 shows Strata's three architectural components, layers, layer repositories and virtual layer file systems. A layer is a distinct self-contained set of files that correspond to a specific set of functionality. Strata classifies layers into three categories: software layers that contain

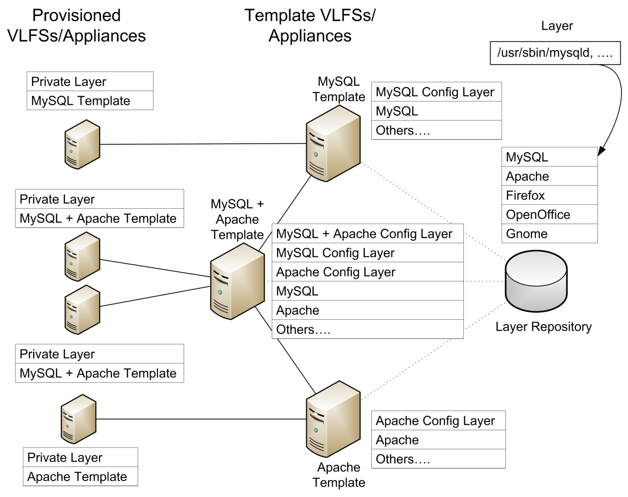

Figure 1: How Layers, Repositories and VLFSs Fit Together

self contained applications and system libraries, configuration layers that contain configuration file changes for a specific VA, and private layers that enable 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 a fully independent entity. They will depend 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 enables a complete fully consistent file system to be automatically provisioned by selecting the main features one desires within the file system.

Layers are provided through layer repositories. As Figure 1 shows, a layer repository is a file system share that contains a set of layers made available to VAs. When an update to a layer 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, for instance due to 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. One can use layers from multiple layer repositories to form a VLFS, as long as the layers are compatible with each other. This enables layers to be provided in a distributed manner. Layers provided by different maintainers can have the same layer names, causing a conflict. However, this is no different though than what exists in 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 that make up a particular VLFS are defined by the VLFS's layer definition file, which enumerates all the layers that will be composed together into a single VLFS instance. To provision a VLFS, an administrator selects a set of software layers that provide the desired functionality and lists those layers in the VLFS's layer definition file.

Within a VLFS, layers are stacked on 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. Software layers, on the other hand, will never obscure files located in a lower layer due to Strata not allowing software layers that contain the same file to be composed together.

4 Strata Usage Model

Strata's usage model is centered around the usage of layers to quickly create VLFSs for VAs as shown in Figure 1. Strata enables 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.

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 is able to turn 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 meta data, including the layer's 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 and the VLFS it will use, (2) determining what repositories are available to it, (3) selecting a set of layers that provide the functionality desired.

For example, to create a template VA that provides a MySQL SQL server, an administrator creates an appliance/VLFS named `sql-server` and select 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 together with a read-write private layer, making the VLFS usable within a VM. The administrator boots the VM and make 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 frozen and becomes the template's configuration layer. As another example, to create an Apache web server appliance, an adminstrator 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. For example, 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, as well as a new configuration layer to contain configuration state that integrates the two services together, for a total of three configuration layers.

4.3 Provisioning Appliances Instances

Given template VAs, VAs can be efficiently and quickly provisioned and deployed by end users by cloning the available templates. 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, (3) forming the VLFS by referencing the already existing respective template VLFS, combining the template's read-only software and configuration layers with a read-write private layer provided by the VM's virtual disk.

4.4 Updating Appliances

Appliances often need to be upgraded to fix security holes in existing software and to add new functionality provided by updated software. Strata upgrades provisioned VAs efficiently using a simple two step process. First, an updated layer is installed into a shared layer repository. Second, the updated layer is incorporated into each provisioned appliance/VLFS.

When an update is installed in a layer repository, all template VAs using that layer in that repository are automatically updated. If the administrator of a template does not desire to upgrade the layer, she can place a hold on it, locking it at its current version.

When a template is updated, all provisioned VAs based on that template will incorporate the update as well. VAs do not have to be powered on to be updated. Since they compose their file system fresh each time they boot, provisioned VAs will compose together whatever updates have been applied to their templates automatically, never leaving the file system in a vulnerable state. Furthermore, running VAs can be upgraded atomically, as Strata can add and remote a layer in a single operation. This is unlike a traditional package management system that when upgrading a package first uninstalls it, before reinstalling the newer version. The traditional method leaves the file system in an inconsistent state for a short period of time due to the possibility of files needed for program execution, such as shared libraries, being unavailable when a program tries to execute due to it not being copied back into place yet.

4.5 Improving Security

Strata makes it much easier to deal with security compromises to the VAs. By dividing a file system into layers, and storing all file system modifications inside the private layer, Strata isolates what changes have been made to the file system from its default configuration. The way to make a compromise persistent is to modify the file system, but Strata makes any file system modifications, be they changing system files or adding a new program to act as a back-door, readily visible in the private layer. This allows Strata to not rely on tools like Tripwire [9] to determine if files have been modified from their default state. This reduces management load due to not requiring any external databases be kept in sync with the file system state as it changes. This segregation of modified file system state also enables quick recovery from a compromised system. By simply replacing the VA's private layer with a fresh private layer, the compromised system is immediately fixed by returning it to its default, freshly provisioned state.

5 VLFS Abstraction

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

Every VLFS is defined by a layer definition file (LDF) that specifies what software layers should be composed together. An LDF is a simple text file that lists the layers and their respective repositories. The syntax of layer lists in the LDF `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, the LDF is modified 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 within the VLFS along with its respective repository. Each layer also has a number indicating the version of the layer that will be composed into the file system. If an updated layer is made available, the LDF is updated to include the new layer version in place of the old one. If the administrator of the VLFS does not desire that a layer be updated, she 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's enables an administrator to only explicitly select the few layers corresponding to the exact functionality desired within the file system, while other layers that need to be part of 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 only has one explicit layer, mysql-server, while it has 21 implicitly selected layers. These include utilities, such as Perl and TCP Wrappers (tcpd), as well as libraries such as OpenSSL (libssl). It also includes a layer providing a shared base common to all VLFSs. Strata distinguishes explicit layers from implicit layers to enable future reconfigurations to remove one implicit layer in favor of another one if dependencies need to change.

When an end user provisions an appliance by cloning a template, an LDF is created for the provisioned VA. Figure 3 shows an example introducing another syntax element, `@`, that instructs Strata to reference another VLFS's LDF as the basis for this VLFS. This enables Strata to clone the referenced VLFS by including its layers within the new VLFS. In this case, as a user just wants 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 presuming 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. 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

```
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: Layer Definition for MySQL server

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

Figure 3: Layer Definition for Provisioned Appliance

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 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 set of functionality. The files are the individual items in the layer that are composed into a larger VLFS. There are no restrictions on the type of files. They can be regular files, symbolic links, hard links or device nodes. A layer can be viewed 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 a 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 enable proper integration of the layer within the VLFS. 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 contained within the layer to be launched automatically when a user wants to play an `mp3` file. Simi-

```
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

larly, if the layer were removed, it should remove the programs contained within itself from the MIME database.

Strata supports four types of configuration scripts, pre-remove, post-remove, pre-install and post-install. When 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 shutdown a daemon, before it actually removed, while a post-remove script can be used to clean up file system droppings that layer leaves in the private layer. Similarly, a pre-install script can make sure the file system is in a manner the layer expects, while the post-install script can start daemons that are included in the layer. The configuration scripts can be written in any scripting language. The layer just has to include the proper dependencies to ensure that the scripting infrastructure is composed into the file system to enable the scripts to run.

Layers are stored on disk, as a directory tree that is named by the layer's name and its 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 that correspond to the layer's three components.

5.2 Dependencies

A key Strata metadata element is its 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 its 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: *dependency*, *pre-dependency*, *conflict*, and *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`), as it dynamically links against the shared library `libwrap.so.0` which is provided by TCP Wrappers. MySQL cannot run without the presence of this shared library, so the layer units that contain MySQL must depend on the presence of a layer unit containing an appropriate version of the shared library. These dependencies constraints can also be versioned to further restrict which layer units satisfy the constraint. For instance, shared libraries can add functionality that breaks their application binary interface (ABI), breaking 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 as 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. This is generally due to an ABI change in the tool chain that builds the shared library. Since the ABI has changed, the new version can no longer satisfy any of the previous dependencies. Therefore, a new layer has to be created with a different name. This ensures that the library with the new ABI is never used to satisfy an old dependency on the original layer. Since the new layer will contain the same files as the old layer, it will have to conflict with the older layer to ensure that they are not integrated into the same file system.

Provide: Provide constraints introduce virtual layers. While a regular layer provides a specific set of files, a virtual layer is an indication that a layer provides a particularly piece of general functionality. Layer units that depend on a certain piece of general functionality being present can depend on a specific virtual layer name in the normal manner, while those layer units that provide that general functionality will explicitly specify that they provide this functionality. For example, many layer units, such as those that provide web-mail or content management software, depend on the presence of a web server, but do not care which one. Instead of depending on a particular web server, they depend on the virtual layer name `httpd`. Similarly, layer units that contain a web server, 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 command. These dependencies in turn depend on the Berkeley DB library and the GNU dbm library amongst others. Due to its dependency mechanism, by just specifying a single layer, Strata is able to automatically resolve all the other layers needed to create a complete file system.

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 enable 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 add an optional version constraint to each layer that is conflicted with to limit the layer units that are actually in conflict. Finally, Depends and Pre-Depends add a boolean OR of multiple layers in their dependency constraints to enable multiple layers to satisfy the dependency.

Resolving Dependencies: To simplify VLFS provisioning by enabling an administrator to only select the explicit layers desired within the VLFS, Strata must automatically resolve dependencies to determine which other layers need to be implicitly included as well. To enable dependency resolution, Strata first provides a database of all the available layer units' metadata as well as the locations of where to reach them. The collection of layer units can be viewed as three sets: the set of layer units themselves, the set of dependency relations for each individual layer unit, and finally the set of conflict relations (C) that define which layer units cannot be integrated into the same file system. This collection can be viewed as directed dependency graph connecting layer units to the layer units they depend on.

A layer unit can be integrated into the VLFS when two principles hold. First, a set of layer units (I), that fulfills total closure of all the dependencies; namely, every layer unit in the set has every dependency filled. Second, $I \times I \cap C = \emptyset$ must hold, meaning that none of the layer units in I can conflict with each other. Determining when these principles hold is a problem that has been shown to be polynomial time reducible to 3-SAT [4, 17], and therefore could be very difficult to solve in the naive manner as 3-SAT is NP Complete. Due to the specialized nature of the problem, an optimized Davis-Putnam SAT solver [5] can be used to solve this efficiently [4].

However, even when a layer unit can be integrated into the VLFS, many times there will be many sets of implicitly selected layer units that allow this. Therefore, Strata has to evaluate which of those sets is the "best". Since Linux distributions already face this problem, tools have been developed to address it, such as Apt [1] and Smart [11]. To leverage Smart, Strata adopts the same metadata database format that Debian uses for packages for its own layers. When Smart is used with a regular Linux distribution, administrators can request it to install or remove packages and it will determine if the operation can succeed and what are the best set of packages to add or remove to achieve that goal. 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 are the best set of layers to add or remove. However, instead of acting directly on the contents of the file system, Strata only has to update the template's VLFS's definition file with the updated set of layers that should be composed into the file system.

5.3 Layer Creation

Strata allows layers to be created in two ways. First, Strata enables `.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 then use. Strata converts packages into layers in a two stage process. 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 and then configure, Strata composes the layers together, and uses the distribution's tools as if the packages they expect to configure have already been unpacked. While Strata is able to convert packages from different Linux distributions, it cannot mix and match them as 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 installs a set of files, they copy it into the correct places in the file system, treating 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 she compiled on the local machine. Much like in package management systems, in Strata, the process of layer creation is instead 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 its own specific directory tree. Namely, the layer creator creates a blank directory to hold a newly created file system tree that will be 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 has to extract programs that integrate the files into the underlying file system and create scripts that can run when the layer is added and removed from the file system. Examples of this include integrating with Gnome's GConf configuration system, the creation of encryption keys, or the creation of new local users and groups for new services that are being added. This leverages skills that package maintainers, in a traditional package management world, need to have as their packages need to be integrated into a traditional file system in a similar manner.

Finally, the layer maintainer needs to setup the metadata correctly. Some elements of the metadata, such as the name of the layer and its version, are simple to set. On the other hand, dependency information can be much harder. However, as package management tools have had to address this issue, Strata is able to leverage the tools that they have built. These tools help the layer creator classify dependencies into two distinct types. The first type is something that can be derived from the executable, while the second is something that has to be known by the maintainer. An example of the latter is a web-mail software layer would need a web server to operate correctly. Based on files in a layer, its difficult to determine that a web server is required. Therefore, the layer creator would have to explicitly list this as a dependency. In practice, every layer that provides basic web server functionality, would contain a provide dependency for the web server virtual layer. Every layer that then needs a basic web server would then be able to depend on this virtual layer name.

On the other hand, many important dependencies can be inferred and Strata can leverage the same tools used to solve this for package management [6]. Consider dependencies due to the executables in the layer dynamically linking against shared libraries. Every executable contains a list of shared libraries that it dynamically links against at run time. While Strata can enumerate this list, many of the shared libraries that it links against might require that a specific version of the library be included depending on what version the executable was built with. To infer this information correctly, Strata has each layer that contains shared libraries include a `shared-libs` file that defines what layer dependencies should exist for programs that are built against it. This enables a layer creator to take an existing Strata system, create a layer within it, and automatically enumerate all the required shared library dependencies of the new layer. If a shared library is unable to be resolved, the maintainer of the layer unit will have to handle that dependency manually. In general, this is due to the layer unit containing the shared library not including an appropriate `shared-lib` file and is considered a bug.

This last problem illustrates how dependency enumeration can be difficult. While we can create and leverage tools to automate certain tasks and make the life of a layer maintainer easier, it still requires problem domain specific knowledge. For instance, many other types of dependencies or conflicts can not be determined automatically and it requires experienced layer maintainers to define them appropriately. In our previous example, a layer providing a web-mail software package will be worthless without a web server. Similarly, a layer that includes Python or Perl scripts will require the presence of an appropriate version of the Python or Perl interpreters. A specific version of the scripting language may be needed, which cannot be easily determined without extensive parsing of the program for specific language features. While problem domain knowledge is required to setup layer dependency metadata correctly, in practice, the large majority of dependency entries (and for many layers, all of their entries) are related to required dynamically linked libraries and can be automated.

5.4 Layer Repositories

Strata provides for two types of layer repositories, local and remote. Local layer repositories are provided by locally accessible file system shares and contain layer units that can be composed into the VLFS. As each individual layer unit is stored as its own directory, a local layer repository contains a set of directories, each corresponding to a layer unit. A local layer repository's contents are enumerated via a database file that provides a flat representation of the metadata of all the layer units present in the repository. The database file is used for providing a list of what layers can be installed, as well as providing dependency information to enable dependency resolution. Local layer repositories can be stored locally within the machine, but more commonly would be stored on a distributed file system such as NFS or on a SAN. By storing the shared layer repository on NFS or a SAN, Strata enables layers to be shared securely among appliances of different users. Even if the machine hosting the VLFS is compromised, the read-only layers will stay secure as NFS or 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 are able to be mirrored into a local layer repository. Instead of having to mirror the entire remote repository, Strata enables an on demand mirroring to occur, where all the layers provided by the remote repository are accessible to the VAs, but have to be mirrored to the local mirror before they can be composed into a VLFS. This enables administrators to only store the layers that they need to use, while maintaining access to all the layers and their updates that the repository provides. Administrators can also specify filters on what layers should be made available to prevent end users from provisioning with layers that violate administration policy. In general, administrator will use these remote layer repositories to provide the majority of the layers, much like administrators would use a publicly managed package repository from a regular Linux distribution.

Layer repositories enable Strata to 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 might want to restrict certain layers to certain end users due to licensing or security issues. Second, as enterprises get larger, they gain more levels of administration. Strata must support the creation of an enterprise-wide policy, while also enabling small organizations within the enterprise to provide more localized administration. Finally, larger enterprises will support multiple different operating systems, and will therefore not be able to rely on a single repository of layers due to inherent incompatibilities between operating systems.

By enabling a VLFS to make use of multiple repositories, Strata solves these three problems. First, multiple repositories enable administrators to compartmentalize layers according to the needs of their end users. By only providing end users with access to the repositories they need to use, they restrict their end users from using the other layers. Second, by enabling sub-organizations to setup their own repositories, Strata enables a sub-organization's administrator to provide the layers that end users need, without requiring intervention of the administrators of the global repositories. Finally, multiple repositories enable Strata to support multiple distinct operating systems, as each distinct operating system can have its own set of layer repositories.

6 VLFS Implementation

To support the creation of 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, as users expect to be able to interact with the VLFS as a normal file system, such as by creating and modifying files, Strata has to enable VLFSs to be fully modifiable. Relatedly, the third problem Strata has to solve is that end users should also be able to delete files that exist on the read-only layer. However, end users should also be able to recover the deleted files by reinstalling or upgrading the layer that contains the deleted. This is equivalent to deleting a file from a traditional monolithic file system, but reinstalling the package that contains the file to recover it. Finally, Strata has to support the ability to dynamically add and remove layers without taking the file system offline. This is equivalent to installing, removing or upgrading a software package that can be done while a monolithic file system is online.

To solve these problems, Strata leverages and expands upon unioning file systems. Unioning file systems enable Strata to solve the first problem as they 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, when two directories are 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`. 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 a file that already existed is modified, the unioning file system change the underlying file directly, in whatever layer of the union it existed previously.

To solve the second problem, union file system can be extended [19] to enable them to assign properties to the 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, a writable 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 up* to the top most layer before the modification occurs. However, since the entire file has to be copied, performance can suffer if this operation has to occur often on large files. This enables VLFS to include a private read-write layer in addition to the set of shared read-only layers that make up the bulk of the file system. When a file is created or modified, it is written to the private read-write layer enabling VLFSs to be differentiated through file system changes.

This layering model also provides a semantic that directory entries located at higher layers in the stack obscure the equivalent directory entries at lower levels. Continuing the example, both layers now contain the file `bar`, but only the top most layer's version of the file is visible. 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. Now, if the file `bar` were deleted, it would not allow the `bar` on the lower layer to be revealed. The white-out mechanism enables obscuring files on the read only lower layers, by just creating the white-out file on the topmost layer.

However, this creates the third problem where a file deleted from a read-only share will never be able to reappear. Unlike a traditional file system, where deleted system file can be recovered by simply reinstalling the package that provided that file, in Strata, white-outs that exist in the private layer will persist and continue to obscure the file even if the layer is replaced. To solve this problem, Strata provides a VLFS with additional private writable layers associated to each shared read-only layer in the VLFS. Instead of containing file data, such as the top-most private writable layer, these layers just contain white-out marks that will obscure files contained within their associated read-only layer. This enables a user to delete a file located in a shared read-only layer, but to only have the deletion persist for the lifetime of that layer's usage. When a layer is replaced, due to reinstalling or upgrading the layer, an empty white-out layer will be associated with the replacement, thereby removing any preexisting white-out. Similarly, Strata has to handle the case where a file belonging to a shared read-only layer was modified and therefore copied up to the VLFS's private read-write layer. Strata provides a *revert* command that enables the owner of a file that has been modified to revert the file's state to its original pristine state. While a regular VLFS *unlink* operation would remove the modified file from the private layer and create a white-out mark to obscure the original file, *revert* only removes the copy in the private layer thereby revealing the original copy below it.

The final problem Strata has to solve is to enable a VLFS to be managed while being used. In a traditional monolithic file system, an administrator can remove a package containing files in use, as deleting a file does not remove its contents from the file system until the file is no longer in use. However, if a layer is removed from a union, the data is effectively removed as well as unions only operate on file system namespaces and not on the date the underlying files contain. If an administrator wanted to modify the VLFS by removing a layer due to deletion or upgrade maintenance, one would be forced to perform the maintenance off-line due to not being able to remove layers that are in use.

To solve this problem, Strata emulates what the `unlink` operation does on a single files and applies it to layer removal. `Unlink` operates in two steps. It first deletes the file name from the file system's namespace, while only freeing up the space taken up by the file's contents when its no longer in use. Traditional package management systems rely on this semantic to enable them to upgrade packages, even if files are in use, by unlinking and then recreating them instead of directly overwriting the files. Strata applies this 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. While this layer is no longer part of the file system namespace and therefore can not be used by any operations that work on the file system namespace, such as `open`, it remains part of the VLFS enabling data operations, such as `read` and `write`, to continue to work correctly for files that were previously opened.

7 Experimental Results

We have implemented Strata as a loadable kernel module on the Linux 2.6 series kernel. Strata requires no changes to the Linux kernel. We present some experimental results using our Linux prototype on various VAs to demonstrate 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 bust adapter connected to an IBM ESS Shark SAN with 1 TB of disk space. The blades were connected by a gigabit Ethernet switch.

To measure management costs, we quantify the time it takes for two common system administrative tasks, provisioning VAs, and updating VAs. To measure overhead, we quantify the storage costs for provisioning many VAs and the performance overhead for running various benchmarks using the VAs. We ran on 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. To provide a basis for comparison, we provisioned these VAs using (1) the normal VMware virtualization infrastructure and Debian package management tools (plain), and (2) Strata. To provide a conservative comparison with 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, and all the other VAs used a 1 GB virtual disk.

7.1 Reducing Provisioning Times

Table 1 shows how long it takes to provision VAs using Strata versus traditional methods using VM cloning. For provisioning 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

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

Table 1: VA Provisioning Times

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 and mount the VLFS with the appropriate layers and continues execution from within it. For provisioning a plain VA using VM cloning, VM cloning copies a template VM to create an instance of the VA.

Our measurements for all five VAs show that traditional methods take much longer to provision VAs than Strata. Provisioning plain VAs takes anywhere from 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 and layers do not need to be copied, so copying overhead is essentially zero. Strata provides roughly five orders of magnitude faster provisioning times than traditional methods, dramatically reducing the management costs associated with provisioning VAs.

The high provisioning costs when using traditional methods would also be incurred when creating new VAs since a new appliance is often created by cloning another appliance to serve as its base without the hassle of setting those components the same way from scratch. In contrast, Strata only has to create a new layer definition file to create a new VA. Creating VAs from scratch using a local layer repository took only .4 s for the Apache, MySQL, Samba, and SSH VAs, and 2.3 s for the Desktop VA. Even if we do not account for the time for modifying a VA to install new packages, Strata provides well over two orders of magnitude faster creation times than using traditional methods for creating VAs.

7.2 Reducing Update Times

Table 2 shows how long it takes to update VAs using Strata versus traditional methods using package management. We provisioned ten VA instances of Apache, MySQL, Samba, SSH, and Desktop for a total of 50 provisioned VAs. Each VA was kept in a suspended state. We then updated the VAs when a security patch was made available for the tar package installed in all the VAs to fix an important vulnerability [18]. For updating the VAs using Strata, 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, it will automatically check to see if the layer definition file got updated and update the VLFS namespace view, an operation that is measured in microseconds. For updating the plain VAs using normal package management tools, each VA instance needs to be resumed, get on the network, have an admin or script ssh into the VA to apply the update and fetch and install the update packages from a local Debian mirror, and finally resuspend the VA.

	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

Table 2 shows the total average time to update each VA using traditional methods versus Strata. We decompose the update time into the time to resume the VM, the time to get access to the network, the time to actually perform the update, and the time to resuspend the VA. The measurements show that the cost of performing an update is dominated by the management overhead of getting the VAs ready to be updated and not the update itself, which is itself dominated by the cost of getting an IP address and becoming accessible on a busy network. The average time to update each plain VA is over 73 seconds. In contrast, Strata does not need to resume the VAs to update them and takes only a second to update each VA. Strata provides over 70 times faster update times than traditional package management when managing just a modest number of VAs. Strata's ability to improve update times would only increase as the number of VAs being managed increases.

7.3 Reducing Storage Costs

Figure 6 shows the total storage space required for different numbers of VAs using plain VAs versus Strata. Using plain VAs, we show the total storage space for 1 Desktop VA, 5 VAs corresponding to an Apache, MySQL, Samba, SSH, and Desktop VA, and 50 VAs corresponding to ten instances of each of the five VAs. As expected, the total storage space required grows linearly with the number of VA instances. In contrast, the total storage space using Strata is almost entirely the space required for the layer repository and is independent of the number of VA instances. For one VA, the storage space required for a single plain VA is less than the storage space required for Strata since 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. However, for larger numbers of VAs, Strata provides a substantial reduction in storage space required since many VAs share layers and those VAs do not need to duplicate the storage requirements for those layers. For 50 VAs, Strata reduces

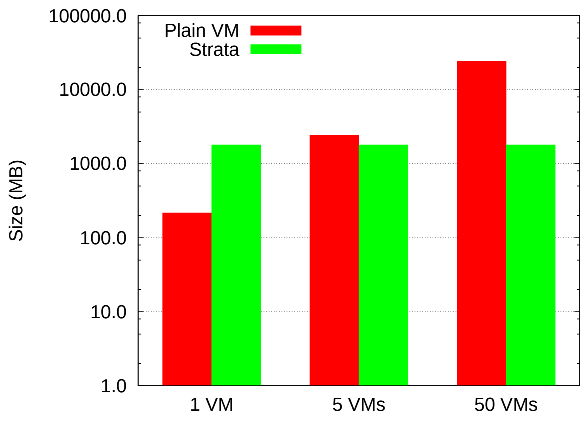

Figure 6: Storage Overhead

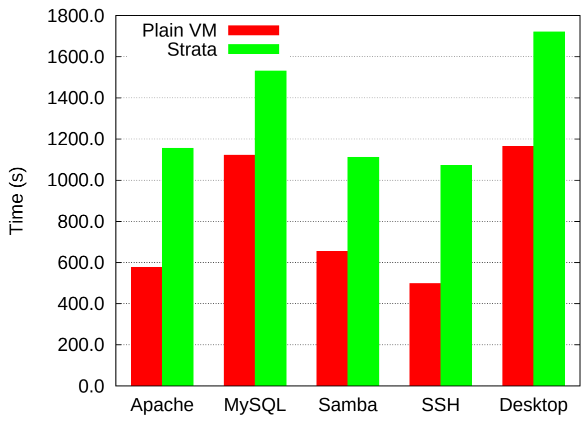

Figure 7: Postmark Overhead

the storage space required by an order of magnitude. Table 3 shows that there is much duplication between virtual machines that are provisioned statically, as the layer repository of 405 distinct layers needed to build the different VLFSs for multiple distinct services, is basically the same size as the largest service.

7.4 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 and compared the results against vanilla Linux systems within a virtual machine. The virtual machine's local file system was formated with the `Ext3` file system, while it was provided read-only access to a SAN partition formatted with `Ext3` as well. We performed all the benchmarks in every scenario described above.

To demonstrate the affect Strata's VLFS has on system performance, we performed a number of benchmarks. Postmark [8], 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 and performed 200,000 transactions. Postmark is very intensive on a few specific file system operations, such as `lookup()`, `create()` and `unlink()` as 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 with Strata. This is because as Strata composes multiple file system namespaces together places significant overhead on namespace operations such as `lookup()`.

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

To demonstrate that postmark's results are not indicative of performance in real life scenarios, 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 to use the two CPUs allocated to the VM. In all scenarios, we added the layers required to build a kernel to the layers needed to provide the service, generally adding 8 additional layers to each case. Figure 8 shows that while Strata imposes a slight overhead on the kernel build compared to the underlying file system it used, it is relatively negligible, under 5% in the worst case.

The second benchmark placed a load on the Apache web server and measured the amount of HTTP transactions able to be completed per second. We imported the database of a popular guitar tab search engine and used the `http_load` [14] benchmark to continuously perform a set of 20 search queries on the database. We measured how many were completed in a 60 s period. For each case that did not already contain Apache, we added the appropriate layers to the layer definition file to make Apache available. Figure 9 shows that Strata imposes a negligible overhead of only 5%.

8 Conclusions and Future Work

Strata's enables system administrators to improve the way they manage the VAs under their control by introducing the virtual layered file system. The VLFS combines traditional package management techniques with unioning file systems in a novel way to provide powerful new functionality. By addressing its contents by file location instead of block address, VLFSs enables Strata to quickly and simply provision VAs, as no data needs to be copied into place. It also provides improved management, as file system modifications are isolated and upgrades can be stored centrally and applied atomically. It also enables Strata to create new VLFSs and VAs by composing together smaller base VLFSs and VAs that

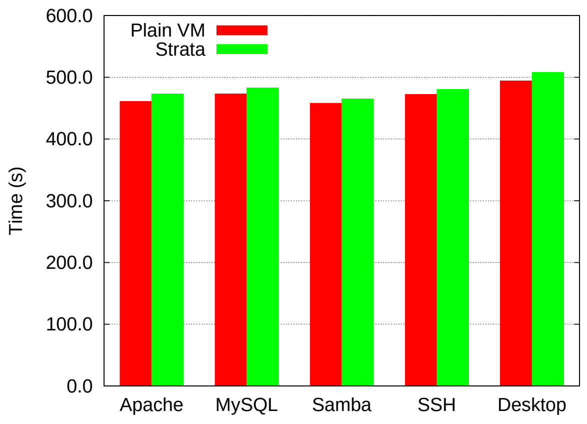

Figure 8: Kernel Build Overhead

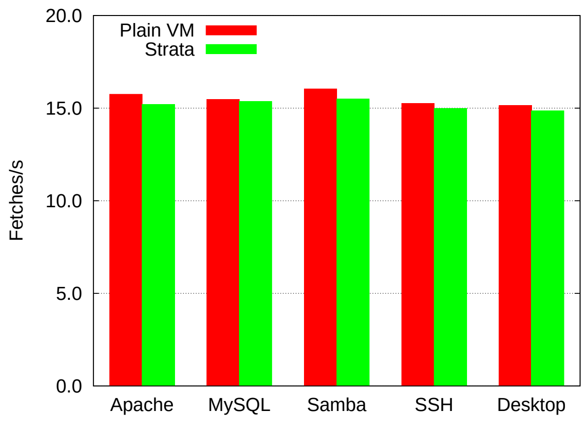

Figure 9: Apache Overhead

provide core components. We have implemented Strata on Linux without requiring any operating system kernel changes, and have demonstrated how a VLFS can be used in real life situations to improve the ability of system administrators to perform their jobs. Strata significantly reducing the amount of disk space required for multiple VAs, allows them to be provisioned almost instantaneously, and enables them to quickly updated no matter how many are in use.

Strata raises a number of follow-up research questions as it only explores the benefits of layers to provisioning and maintaining a file system. First, as layers can be used to branch a file system state into two distinct entities, what benefits can that bring to the administration and testing of systems? Second, dividing up multiple applications amongst multiple machines can keep the rest of the applications' data secure, if one is compromised. However, having to use multiple machines to access individual applications is not a very usable environment. Therefore, how can Strata be leveraged within a desktop environment to provide a more secure desktop, while retaining the general desktop experience users expect.

References

[1] B. Byfield. An apt-get Primer. `http://www.linux.com/articles/40745`, Dec 2004.

[2] 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 *21st Large Installation System Administration Conference*, Dallas, TX, Nov 2007.

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

[4] R. D. Cosmo, B. Durak, X. Leroy, F. Mancinelli, and J. Vouillon. Maintaining Large Software Distributions: New Challenges from the FOSS Era. *EASST Newsletter*, 12:7–20, 2006.

[5] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[6] Debian Project. DDP developers' manuals. `http://www.debian.org/doc/devel-manuals`.

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

[9] G. H. Kim and E. H. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. In *In Systems Administration, Networking and Security Conference III. Usenix*, 1994.

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

[11] G. Niemeyer. Smart PM. `http://labix.org/smart`.

[12] J.-S. Pendry and M. K. McKusick. Union Mounts in 4.4BSD-lite. In *USENIX 1995 Technical Conference*, 1995.

[13] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *3rd Symposium of Networked Systems Design and Implementation*, May 2006.

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

[15] 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 *2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Mar 2008.

[16] Rpm package manager. `http://www.rpm.org/`.

[17] A. Towns. Checking Installability is an NP-Complete Problem. `http://lists.debian.org/debian-newmaint/2007/11/msg00023.html`.

[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.