"*The first project was to shorten discourse by cutting poly-syllables into one, and leaving out verbs and particles, because in reality all things imaginable are but nouns.*"
- Jonathan Swift, Gulliver's Travels

University of Alberta

Slimming Virtual Machines based on Filesystem Profile Data

by

Jeremy James Nickurak

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2010
Edmonton, Alberta

Examining Committee

Paul Lu, Computing Science

Ioanis Nikolaidis, Computing Science

Warren Gallin, Biological Sciences

Abstract

Virtual machines (VMs) are useful mechanisms for better resource utilization, support for special software configurations, and the movement of packaged software across systems. Exploiting VM advantages in a production setting, however, often requires computer systems with the smallest possible disk-size footprint. Administrators and programmers who create VMs, however, may need a robust set of tools for development. This introduces an important conflict: *Minimalism* demands that packaged software be as small as possible, while *completeness* demands that nothing required is missing. We present a system called *Lilliputia*, which combines resource usage monitoring (through a Linux FUSE filesystem we created called *StatFS*), with a filtered *cloning* system, which copies an existing physical or virtual machine into a smaller clone. Finally, we show how Lilliputia can reduce the size of the *Trellis Network-Attached-Storage (NAS) Bridge Appliance* and the *Chemical Shift to 3D Structure* protein structure predictor to 10-30% of their original size.

Acknowledgements

The production of this dissertation, the implementation of its components, and the evaluation of its effectiveness would not have been possible without the support of many, and I must make clear my appreciation to them.

First and foremost, to my loving family, for unending patience, support and motivation.

To my fiance Julie, to my lab mates Cam, Erkan, and Jordon, to everyone with the Undergraduate Association of Computing Science and the Computing Science Graduate Student Association at the University of Alberta, to Edward and Jon, my partners in the USRA research that led to this work, and to all my other friends I don't have room to mention here: your companionship, good humour, and advice were invaluable.

To my supervisor, Paul, your patient and encouraging example is a great inspiration.

For financial assistance, I would be remiss if I did not acknowledge support I've received from the Alberta Prion Research Institute (APRI) and the Natural Sciences and Engineering Research Council (NSERC).

Finally, I must also give thanks and praise to God, without whom nothing, not the least of which my search for "whatsoever things are true", would be possible.

Contents

List of Tables

List of Figures

List of Acronyms and Abbreviations

Acronym	Definition and location of first use
AFS	Andrew File System, Subsection 2.1.5
CD	Compact Disc, Section 1.1
CGI	Common Gateway Interface, Subsection 4.1.2
CIFS	Common Internet Filesystem, Section 1.2
CPU	Central Processing Unit, Subsection 2.1.1
CS23D	Chemical Shift To 3D Structure, Section 1.1
DVD	Digital Versatile Disc, Section 1.1
FUSE	Filesystem in User Space, Subsection 2.1.5
HTTP	Hypertext Transport Protocol, Subsection 4.1.2
ISR	Internet Suspend/Resume, Subsection 2.2.3
JVM	Java Virtual Machine, Subsection 2.1.2
KVM	Kernel Virtual Machine, Section 1.4
LDAP	Lightweight Directory Access Protocol, Section 1.4
LRP	Linux Router Project, Subsection 2.2.2
MBR	Master Boot Record, Subsection 3.3.1
NAS	Network-Attached Storage, Section 1.2
NFS	Network Filesystem, Section 1.1
NIS	Network Information System, Section 1.4

NTFS	NT Filesystem, Subsection 2.1.4
OS	Operating System, Section 1.1
P2V	Physical to Virtual, Section 1.1
PCI	Peripheral Component Interconnect, Subsection 2.1.2
PID	Process ID, Subsection 3.3.5
POSIX	Portable Operating System for unIX, Subsection 2.1.5
PPP	Point-to-Point Protocol, Subsection 3.1.3
RAM	Random Access Memory, Subsection 2.1.5
SFTP	Secure File Transfer Protocol, Section 4.3
SMB	Server Message Block, Section 1.2
SQL	Structured Query Language, Subsection 3.3.6
SSH	Secure Shell, Section 1.2
TNBA	Trellis Network-Attached-Storage (NAS) Bridge Appliance, Section 1.2
USB	Universal Serial Bus, Subsection 3.1.1
VA	Virtual Appliance, Section 1.1
VAA	Virtual Application Appliance, Subsection 2.2.2
VFS	Virtual Filesystem, Subsection 2.1.5
VM	Virtual Machine, Section 1.1
WAN	Wide Area Network, Subsection 3.1.1

Listings

Chapter 1

Introduction

1.1 Motivation

Virtual machines (VMs) are a useful mechanism to address many issues in software development and deployment: software encapsulation, supporting legacy systems, server consolidation, and cloud computing. A VM can encapsulate the entire *identity* of a computing system, from operating system (OS) to libraries and applications. An encapsulated machine provides a number of benefits:

1. Multiple encapsulated machines can be deployed on a single physical server. Consolidating these machines in one place can improve resource utilization (via statistical multiplexing) [3], decrease power consumption, and save physical space in server rooms and racks. For example, many server environments will include print servers, file servers, web servers and more, all configured and tuned from the OS-up for their particular tasks. Each could be running a different set of software, different versions of libraries, and incompatible operating systems. With virtualization, each one of these servers can have its own isolated computing environment (with no more interaction with each other than in the regular physical networked case), but all running on a single piece of physical computing infrastructure.

2. Legacy operating system installations can be maintained in an encapsulated machine to support applications with specific needs, while the operating system *hosting* those installations remains up-to-date. For example, a web-facing server running older software incompatible with modern operating systems could present a security liability. By configuring a host machine with an up-to-date operating system and software suite, and passing only required requests in to a VM running the legacy software, all the services can still be provided, while mitigating the security implications of

running older code.

3. Encapsulated machines can be packaged and moved from place to place as a self-contained unit, allowing duplicate deployments, or even mobile deployments that migrate on demand. For example, an entire server designed to run a special-purpose application can be packaged in a VM, and made available for download. Additionally, long-running software inside a VM can be suspended, moved, and resumed in a new location in response to outages or changing server resource demands.

In particular, the notion of creating a VM-based virtual appliance (VA), a purpose-specific VM, has garnered significant attention recently [50]. A VA can potentially be customized, tuned, and made smaller in size if it does not have to support general-purpose use. Note that a VA is just a specific kind of a VM, therefore we also use the term VM to refer to VAs.

The basic process of taking an existing physical server and cloning it into a VM (i.e., *Physical-to-Virtual* (P2V)) is now commonplace, with both commercial (e.g., VMware vCenter Converter [49]) and other solutions (e.g., using `dd` or `tar` in custom scripts, similar to Listing 3.1 in Subsection 3.3.1). Copying the local disk(s) of the physical server onto the VM disk image of the VM *encapsulates* a *full-sized* version of the server into a VM image. We say that the "output" VA encapsulates the "input" machine if it retains all the same desired behavior of the original. For example, the disparate file servers, print servers, and web servers from point 1 (above) can be converted one-by-one into VMs, and then deployed to one physical machine. After some tweaks of the boot process and other internals of the VM, the full-sized, virtualized server is ready. Likewise, creating a VM from scratch (such as for a VA) via the original OS installation CD/DVD media (e.g., Windows, Ubuntu Linux, Ubuntu JeOS) is also commonplace. In either case, the output VM is functionally equivalent to the input physical machine, or any other machine with the same set of software and data, with all the advantages of a VM described in Subsection 2.1.3.

Whether created via P2V or from scratch, and whether configured as a traditional server or as a VA, the resulting VM images are often several gigabytes in size. For example, the Chemical Shift to 3D Structure (CS23D) deployment (a bioinformatics server described below) used throughout this project was over 30 gigabytes when provided to us, as in Table 1.1. Naturally, removing unnecessary resources from these VAs has benefits whenever these images are backed-up, migrated, copied, provisioned, and when installed on live CDs/DVDs, portable drives, or smaller devices, such as netbooks. Creating and maintaining minimal VAs, however, presents specific practical problems when it comes to *completeness*,

including all the necessary files, and *minimalism*, excluding all of the unnecessary files.

For example, the basic P2V process is complicated when the physical server is itself not self-contained and uses, for example, network resources such as remotely mounted filesystems via the Network File System (NFS). A naive approach to the P2V conversion of a server with NFS mounts will either omit the remote NFS volumes (which would leave the VA incomplete), or include *all* remote NFS volumes (which would dramatically increase the size of the VA), or require that the VA have the ability to mount the remote NFS volumes (which might reduce the mobility of the VA and/or create NFS security issues).

Furthermore, there might be files on both local and remote disks that are not required for the production VM. For example, many servers have documentation files and development environments with compilers, integrated development environments (IDEs), multiple scripting languages, and a full suite of libraries and header files. For production use, many of these files are not necessary.

Manually selecting which files or volumes (e.g., NFS) to encapsulate into the VM via an *include list* for the P2V is possible. Similarly, creating a manual *exclude list* is possible. However, as with manual versus automatic package management (e.g. `apt-get` and `yum`) for software, manually maintaining meta-data about software, like our include and exclude lists, is error-prone and complicated. For example, dependency information stating what libraries are required for a given piece of software may be error-prone.

Furthermore, keeping the original server and/or a full-sized VM image for ongoing development purposes is likely desirable. In other words, the P2V process is not necessarily a one-time event. In fact, creating a *slimmed* production VM from a full-sized development VM might be as common as an over-night rebuild of the system. Therefore, manually updating the include and exclude lists for each production build is likely to become an untenable chore.

The *Lilliputia* system is our practical solution to the problems of completeness and minimalism, by automatically creating the include and exclude lists. Manual, supplementary include and exclude lists are also supported, but the bulk of the effort is automated. Our automatic encapsulation and slimming process is based on a new, simple, user-level filesystem called *StatFS*, which traces and logs the actual resources (both local and network-based) accessed by the running server. Post-processing the StatFS logs automatically generates the include and exclude lists. When in production mode, StatFS is not present at all. Even when tracing a system (which is *not* the common case), the run-time overheads of StatFS are less than 4% in our experiments (Section 4.5). Given all of the software and files that

Virtual Appliance	Full Size	Slimmed Size	% savings
TNBA	591	54	92%
CS23D	31876	9366	71%
TNBA (compressed)	242	27	89%
CS23D (compressed)	9305	2054	78%

Table 1.1: Summary of slimming results for the Trellis Network-Attached-Storage (NAS) Bridge Appliance (TNBA) and CS23D in MB

are installed by default on most servers, and given the large number of NFS volumes that are typically mounted on servers, we have seen slimmed image sizes in the range of 10% to 30% of the size of the full-sized images (e.g., 54 MB versus 591 MB, 9.37 GB versus 31.9 GB, as in Table 1.1).

1.2 Examples

In Section 4.1, we describe a bioinformatics server to use chemical shift data to compute 3D protein structure known as CS23D [53]. The server uses bioinformatics databases that are shared with other servers. Given the need to regularly and consistently update these shared databases (e.g. protein databases), they are NFS mounted (a common practice) and not replicated. Of course, tools for consistent replication are available, but the point is that NFS mounting is common in practice. Furthermore, CS23D was developed by so many different people (some of whom had already left the project, some of them external to the project) and uses so many different components that it would have been impractical to try to manually enumerate all of the network resources used by the system. The large number of developers involved, and the large number of software components used, are typical of many such projects.

After CS23D had been in production for months, it was decided that a VM-based CS23D was desirable. We and other developers wanted the ability to quickly launch a CS23D VM on a different server to maintain uptime (i.e., redundancy, availability), the ability to run multiple CS23D VMs to serve more clients (i.e., capacity, mirrors), and the ability to continue development of CS23D within a VM without affecting the physical CS23D (or the other CS23D VMs).

The full-sized CS23D VM, after installation of all the tools required to develop and maintain the system, was over 30 GB in size. On the one hand, many servers now have terabytes of disk, so the quantity of disk storage is not the issue. On the other hand, many

researchers and developers use laptops that (currently) have much less than a terabyte of disk. Creating each clone of the VM image for a new development branch requires a time-consuming copy of 30 GB. In addition, 30 GB VM images are not practical for Web-based distribution to our research partners and the use of inexpensive, dual-layer DVDs requires the image to be less than 8.54 GB in size. Even the process of backing up tens of gigabytes of VM image files (which may be modified frequently) may consume many resources. Finally, the increasing use of live VM migration [11] may also benefit from decreased initial image size, in implementations where the entire filesystem state has to be moved from point-to-point before the migration can be considered successful.

Fortunately, using Lilliputia, our slimming techniques reduced the CS23D VM to 9.37 GB before compression, and 2.1 GB after compression, as shown in Table 1.1.

As another motivating example, in 2006, our research group created the Trellis Network-Attached Storage (NAS) Bridge Appliance [12] (TNBA) VA for the VMware Ultimate Virtual Appliance Contest (UVAC) [48], winning Second Prize. In brief, the TNBA is a server that presents a network filesystem interface (via SMB/CIFS, or *Server Message Block/Common Internet File System*) to clients on the front-end, but can access files via Secure Shell (SSH) on the back-end. Therefore, files that are not available via a traditional distributed filesystem (e.g., due to administrative domain issues), but can be accessed via SSH, can still be made available via the TNBA to SMB/CIFS clients. The TNBA is discussed in more detail in Subsection 4.1.1.

Notably, one of the criteria for the UVAC was the size of the resulting image, since the bandwidth required to distribute VAs is an understandable concern (as it is for our CS23D VM). An earlier approach to slimming was used with the TNBA. The final, slimmed TNBA VA size of 16 MB (compressed) was certainly a factor in the VA's Second Prize finish. The original slimming technique used in 2006 was based on manipulating filesystem timestamps, which does not work as cleanly in practice with shared, network-mounted filesystems.

In this work, we re-create the *slimmed* TNBA with the new StatFS-based encapsulation and slimming technique. The final, slimmed VM image is 27 MB compressed (54 MB uncompressed), as compared to a full-sized image size of 591 MB, as shown in Table 1.1. Note that the change from 16 MB in 2006 to 27 MB in 2009 is due to a change from Gentoo to Ubuntu Linux for the guest OS installation, and the corresponding increases in package size and software capabilities from 2006 to 2009.

1.3 Contributions

The contributions of this work (modulo the caveats below) are as follows:

1. In Chapter 3, we present the Lilliputia system, which implements trace-based approach to automatically generating include and exclude lists for creating complete and minimal VM images, and clones systems based on these lists.

2. We demonstrate the effectiveness of Lilliputia on two real-world VMs. Chapter 4 presents a case study of each: CS23D and the TNBA. With each VM, both the *fresh disk* and *slimming* approaches were applied. We verify that the slimmed VM images have all the functionality required for production use, but are about 29% and 9.1% the size (uncompressed) of the full VMs, respectively.

1.4 Caveats

Some limitations of our work at this time are:

1. As with software testing, our approach to encapsulation and minimalism for VMs depends on proper trace coverage. StatFS trace runs can be derived from the existing test programs (assuming, optimistically, that test-suites are in place). For corner cases not (currently) covered by the trace runs, supplementary include and exclude lists can be manually specified. In our experiments, these supplementary lists are a few lines for the TNBA case and a manageable page of entries for the CS23D case (for reasons detailed in Subsection 4.5.2). Our goal is to automate the creation of include and exclude lists as much as possible.

2. Another issue in the encapsulation of a server is user account management. Many servers use either the Network Information Service (NIS) or some variation of the Lightweight Directory Access Protocol (LDAP) to manage the user accounts. In short, our encapsulation technique does not help with network-based user account management (just network-based filesystems). All of our VMs to date have used local user accounts, which is consistent with the notion of a VA.

3. Our metric of focus is amount of disk storage required (in megabytes (MB) or gigabytes (GB)) for VM images, when not in use. Another important metric is the resource footprint (both memory and disk) of a running VM. As we note below, VM disk images grow over time as the VM is used, for reasons related to VM disk image

overheads and internal fragmentation. However, resource footprints and growing disk images are heavily dependent on the specific guest OS configurations and workloads. Therefore, we leave these other, important metrics for future work.

4. Of course, this work's contributions are on the practical (as opposed to conceptual or theoretical) side of VM construction and how they can be used on resource-constrained machines. Although our empirical experiments are based on Linux Kernel Virtual Machine (KVM) the basic ideas are portable to other VM hypervisors and, with more work, other guest OSes besides Linux.

1.5 Summary

VM-based appliances have many strengths. Automatically making the VA disk images as small as possible, without sacrificing functionality, makes them more manageable, by making them easier and faster to deploy, back-up, or migrate. Lilliputia, as shown through experimental case studies, displays precisely this ability to reduce image size, producing functional slimmed VAs.

In the following chapters, we will introduce background concepts important to our use of virtualization and filesystems. Mechanisms for virtualization, filesystem concepts, and existing implementations of virtualization *hypervisors*, profiling tools, and similar projects will be discussed. With a good understanding of these technologies, the design and implementation of Lilliputia's trace-based filesystem and slimming system will be presented. Finally, we will discuss our experiences using the complete Lilliputia system to create real-world slimmed VMs with the TNBA and CS23D, and our evaluation of them for effectiveness at reducing image size, and maintaining correctness.

Chapter 2

Background and Related Work

In order to understand the problems of making virtual machines (VMs) as small as possible, it is important to understand several background concepts of virtualization. Additionally, several related research projects provide insight into the mechanisms by which we were able to reduce the size of VMs.

2.1 Background

2.1.1 Motivations for Virtualization

Virtualization, in broad terms, is the approach of running a VM, which has all the behavior and observable appearance of a physical machine, but as a *guest* process executed on another *host* machine. Virtualization may include the ability to run different operating systems (OSs) and different versions of OSs, generally simultaneously, on a single unit of physical computing infrastructure. For example, a guest VM could be running Windows, inside of a Linux-based host machine. The chief benefit of such an approach is isolation of guest behavior from the host. Virtualization allows software in a guest to be upgraded or modified independently of software on the other guests or the host system, isolating them from the effects of one another.

As with historical VMs, modern uses of VMs include allowing different OSes and different versions of OSes to share the same CPUs. Software did not have to be upgraded in unison, as long as a VM could be created with the necessary OS and libraries to run systems along-side other VMs. VMs offer greater flexibility in software packaging and configuration, and remains a powerful use of the technology. For example, a guest VM running a Red Hat distribution of Linux can be updated to a newer version, without impacting a host, running a Debian distribution of Linux. In particular, the guest OS could be updated to a development version without making the host unstable. A guest could even run a completely

different base OS, such as Windows XP.

VMs allow greater flexibility in mobility of a packaged set of software, even allowing an entire VM to be bundled together, moved to another location (possibly many different locations simultaneously), and executed, with little modification. It is also an extension of the time-sharing concept which was so popular with early mainframe computers: by presenting the user with what appears to be a full privately-owned machine, virtualization maximizes the options available to that user, without impacting others.

2.1.2 Emulation to support Virtualization

At one extreme, and not the subject of this dissertation, the goals of virtualization can be achieved through full emulation. In an emulated environment, software simulates the state of a machine, and alters that state iteratively. Since the process of a machine is entirely simulated, it is even possible to execute binary code from a hardware architecture entirely different from the hardware running the code. Assembled instructions and program counters are loaded into emulated registers on a emulated processor, and executed, causing the emulated machine to perform load and store operations on emulated memory, and invoking emulated I/O-calls that interact with emulated I/O controllers, emulated disks, emulated PCI devices, and so on.

A related case where the term *virtual machine* is frequently used is that of the Java Virtual Machine. The Java Virtual Machine (JVM) interprets and executes Java bytecode, assembled instructions which typically are not intended to run on physical hardware. By implementing the instruction set in Java bytecode, a JVM effectively *emulates* the Java platform.

While the emulation approach produces the same output as a real machine for the same input (assuming a deterministic program with no timing sensitivity), it is both infeasibly slow for many purposes, and relatively complicated to implement.

A beneficial modification to the emulation approach is to translate binary code before execution. Guest program text segments can be translated in bulk into equivalent instructions for the host architecture, and executed natively, provided the input and output effects can be mapped back into the simulated machine correctly. The binary translation approach, used in emulators like Qemu [5], as well as so-called "Just-In-Time Compilation" in JVMs, dramatically improves performance over simulating each individual instruction independently, limiting the comparatively hard work of full emulation to I/O device interaction, and other privileged instructions that must be emulated.

2.1.3 Same-instruction-set Virtualization

Figure 2.1 provides an illustration of how components of a VM map to elements in the host physical machine.

When the host physical architecture is the same architecture as that which a VM is designed to run on, most instructions can be executed directly by the physical CPU and memory without any translation at all (as in "Virtualized Instruction Execution" and "Host Memory Region" in Figure 2.1). In cases where direct execution of guest VM code would violate the isolation of the guest and the host from each other (or otherwise behave incorrectly), the offending instructions can be translated ahead-of-time as above. For example, a guest VM running Linux for a 32-bit x86 processor (as an example instruction set architecture) can execute most of its instructions natively on a host running on a physical 32-bit (or, as an aside, even 64-bit) x86 processor. Code in the guest VM which attempts to read or write on a hard disk or other I/O device will be translated to instead emulate the same effect on a virtual device (as in "VM Disk Image" and "Emulated Devices" in Figure 2.1), and then resume native execution of the VM guest code.

Most discussions of the history of VMs begin in the 1960's with IBM's 360 (and related) systems that had hardware support for hypervisors and could multiplex expensive hardware (e.g., mainframes and minicomputers). Since the 1960's, VMs have never really gone away. However, the advent of VM technology for commodity x86 systems (e.g., VMware, Xen, and the Linux Kernel Virtual Machine (KVM)) has made the technology even more accessible.

Two technologies which have made modern virtualization more effective are:

1. **Hardware assisted virtualization of instructions:** With the increasing use of virtualization in typical computing environments, manufacturers became more inclined to design hardware with virtualization in mind. To that end, both Intel and AMD processors now support a series of virtualization extension instructions. VT-x and AMD-V (from Intel and AMD respectively) provide a mechanism to pass guest code to be executed to the CPU, in isolation from the host OS. Many of the privileged instructions that would otherwise have to be emulated can then be trapped and emulated on demand, making the "Virtualized Instruction Execution" phase faster and simpler. Later implementations of these extensions also include virtual page table translation facilities, allowing accelerated translation from guest virtual memory addresses to host physical memory addresses, improving speed of the "Host Memory

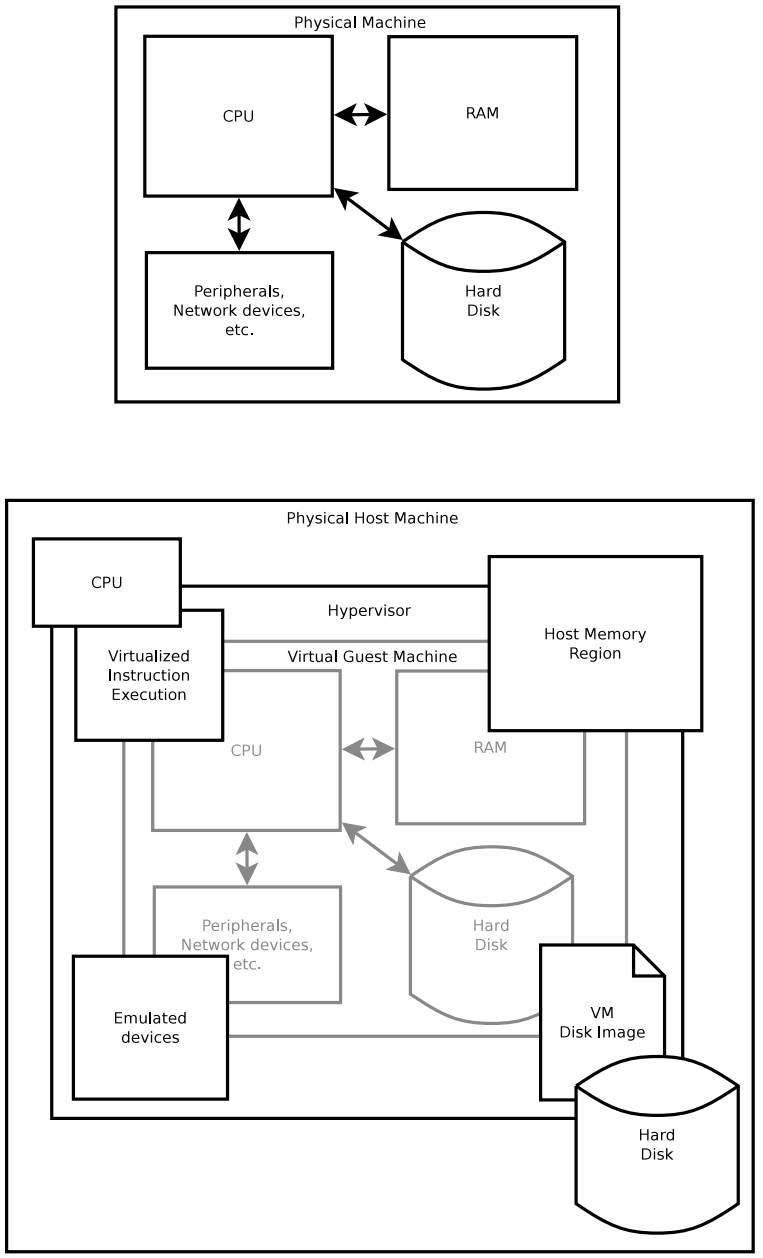

Figure 2.1: Illustration of Virtualization. Above, a traditional physical computer with a CPU, memory, hard disk, and other devices. Below, the same machine as simulated in a VM on another host.

Region" portion of Figure 2.1.

2. **Para-virtualization:** The above techniques of emulation and virtualization implement what is called *full virtualization*. Under full virtualization, the guest virtualized computer is intended to be indistinguishable from a real physical machine. Full virtualization has the benefit that software designed to operate on a real machine will behave just the same under virtualization, because real-world I/O devices are fully emulated. For example, a brand-name network adapter could be emulated to provide a VM with access to a network.

   If, however, the guest OS is aware that it is running under a virtual environment (i.e. "*para*virtualized"), it can provide faster and simpler implementations, tailored to virtualization. For example, the Virtio project [37] provides virtual network and block devices, which are simpler for the host hypervisor to implement. Virtio also provides a corresponding integrated user-space driver. Effective integration between the para-virtualized device and the para-virtualization-aware driver can limit the number of times data must be copied to get from the hypervisor to the guest OS or vice-versa.

A more recent advance in virtualization technology is the opportunity to migrate [11] a running guest VM from one host to another. The simplest approach to migration is to create a complete snapshot of the state of the guest VM's CPU, memory, disk, and other devices. This snapshot can then be transferred to a new location, loaded into another host virtualization environment, and resumed. Later developments in migration enable guest VMs to be migrated live, with minimal downtime.

2.1.4 Virtual Machine Disk Images

With the above virtualization techniques, it is possible to simulate most of the behavior of a traditional physical computer. However, the critical components that distinguish any one computer from another are mostly in its persistent long-term state, typically as a simulated hard disk drive, or *VM disk image*, in the "VM Disk Image" portion of Figure 2.1. On a host computer, a VM disk image is typically a single, ordinary file. Inside a guest, a VM will use standard filesystem formats Linux Extended filesystem or the NT File System (NTFS). These filesystems treat a VM disk image just as they would a real disk, that is, they partition the disk into volumes, allocate metadata structures such as inodes and journals, and partition files into fixed size blocks and store the blocks on disk. Three common implementations of VM disk images are illustrated in Figure 2.2.

The simplest implementation of a VM disk image is a straight one-to-one mapping of guest disk block to host file block, as in 2.2a. Each block in the VM disk image file on the host is allocated ahead-of-time to one block of the VM's simulated block device. Because the file is allocated this way, the size of the disk image is fixed, and will not grow over time. The Qemu hypervisor and supporting utilities refer to this format as *raw*.

Conversely, most virtualization technologies support *growable* disks, as illustrated in 2.2b. With a growable disk, blocks are only written to the host VM disk image file when they're written to inside the guest. This way, a simulated 30gigabyte disk which only contains, for example, 500 megabytes of data, doesn't necessarily need to consume 30gigabytes of the host's storage. As the VM disk image is used, files will be created and deleted, but VM disk images are not automatically defragmented, so they only grow monotonically, even though empty space may exist within the allocated portion. For example, the growable VM disk image for the Chemical Shift to 3D Structure (CS23D) server was over 40 GB even though the `du` utility reported approximately 30 GB of files on the disk in the VM (see Table 4.2 below). Eliminating wasted space after a VM has been created is important if a VM is to remain portable in size.

Finally, the *linked clone* option, illustrated in 2.2c, provides a feature similar to version control software, but in a VM disk image context. A VM disk image file can be used as a *parent* image for a new *child* disk. With the linked clone feature, a VM configured to use the child disk will be able to read any blocks present in the parent, if those blocks are not present in the child. If data is overridden in a child disk, a *copy-on-write* feature brings a copy of the parent block into the child, where it can be modified there without affecting the parent. This feature can allow the parent image to act as a historical *snapshot* of what a VM looked like at one point in time, while still allowing modifications in the child. It also allows the sharing of VM disk images. For example, a base Linux distribution can be installed in a parent image, common to multiple VMs. Each VM can then have their own child image, which is a descendant of the common parent, and can have separate configuration and application data there.

2.1.5 Filesystems as Interfaces

As the permanent mass storage of a VM is such a critical element to its behavior and implementation, it is important to understand the impact of filesystems layered on top of them. Filesystems provide a meaningful view to the data on a large block device, partitioning the blocks into individual files, organizing them into a directory structure, and implementing

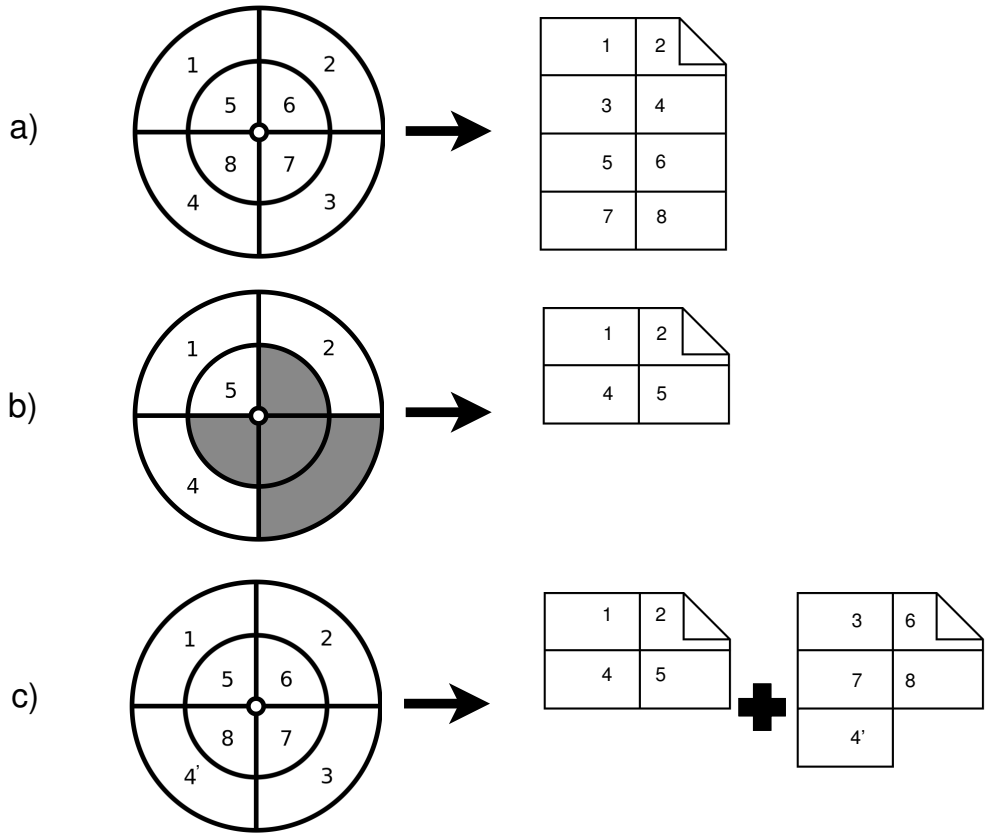

Figure 2.2: Illustration of VM disk image formats. (a) "Raw" non-growable format, wherein every block in the guest virtual disk is mapped directly to a block in the VM disk image file. (b) Growable format, wherein only blocks which have been written to in the guest virtual disk is actually written to the VM disk image file. Here, grey indicates unwritten guest disk region. (c) "Linked clone" copy-on-write format, wherein blocks 1, 2, 4, and 5 reside on a read-only parent VM disk image file, and blocks 3, 4', 6, 7, and 8 reside in a separate read-write file. In this case, block 4' is a modified version of block 4, which necessitates a fresh copy of that block into the clone disk.

metadata like file names, ownership, and permission settings.

Traditional UNIX filesystems are designed with permanent storage (for example, a rotating-disk hard drive) in mind. Optimizations are made to put data which is likely to be accessed with close temporal locality in close physical locality, for example, on the same cylinder on multiple storage platters. Likewise, limiting fragmentation, wherein data for a particular resource is spread out between many places on the disk, is a major priority. These concerns become less relevant as higher-level abstractions are layered on top of the storage infrastructure. Network filesystems like the Andrew File System (AFS) and Network File System (NFS) store actual file data remotely, where it can be difficult or impossible to make optimal storage decisions.

More recently, filesystem concepts are being applied in areas that are even further removed from the idea of a permanent mass-storage device. Modern pseudo-filesystems, which generally do not involve permanent storage, act more as a POSIX-influenced interface to other applications. Linux systems, for example, use `proc` and `sys` filesystems to provide a POSIX interface to kernel and driver internals. Reading the `/proc/cpuinfo` will return kernel-provided hardware information, while writing "mem" or "disk" to `/sys/power/state` will cause a machine to switch to one of 2 low-power modes (suspend-to-RAM or hibernate-to-disk, respectively).

Linux's *Filesystem in Userspace* (FUSE) interface is one option for implementing both real filesystems and pseudo-filesystems. With FUSE, read/write calls (and all other filesystem operations) to the Linux Virtual File System (VFS) layer are translated into subroutine calls to a userspace binary program (as opposed to a kernel-module for servicing filesystem requests). As illustrated later in Figure 3.2, the VFS is a standard in-kernel interface to receive and dispatch filesystem requests. The userspace program that receives redirected VFS calls then implements the logic that actually retrieves or stores information in the filesystem data and metadata. By virtue of being implemented in userspace, these filesystem modules have access to a wide range of libraries, programming languages, and scripting language interpreters, bringing an especially rich set of potential features to a POSIX interface that is usually limited by kernel facilities.

As a result, a wide selection of FUSE filesystem implementations have become available, from implementation of filesystems from other OSs (such as the zfs-fuse [15] and NTFS-3g [45] projects), distributed cluster filesystems (such as GlusterFS [17] and MooseFS [6]), to more esoteric pseudo-filesystems like BloggerFS [38], which provides a POSIX filesystem interface to read and write online journal-entries with a standard text

editor. Isolating FUSE filesystems in userspace means their effects on the rest of the system are mitigated, meaning highly experimental features can be implemented without risk of damage to other running systems. FUSE suffers, however, from a performance penalty as a result of increased content-switching between the userspace filesystem implementation and the kernel's VFS layers. This is discussed in more detail in Subsection 3.3.4

2.2 Related Work

2.2.1 Virtualization Hypervisors

One of the best known commercial virtualization products is VMware [51]. VMware's family of virtualization software, available for a number of x86-based platforms, historically used binary translation only where necessary to provide fast VM execution. Later versions also take advantage of hardware virtualization support.

Xen [4] and KVM [36, 23] are the leading open-source virtualization infrastructures. Xen provides a small hypervisor OS that performs device emulation, on top of which separate VM's are executed. KVM, by contrast, implements virtualization within a Linux kernel module. VMs then execute as a traditional Linux process.

Qemu [5] is one piece of software that takes advantage of the Linux KVM infrastructure. Qemu is historically a full emulator, and still supports emulation of many different hardware architectures on top of Linux. As a result of this heritage, Qemu has excellent support for I/O device emulation. Coupled with KVM support, Qemu-KVM [19] is fast, easy to use, and an excellent platform for development, and was thus used as the platform for Lilliputia (although Lilliputia has no specific ties to any particular hypervisor).

2.2.2 Virtual Appliances and Virtual Application Appliances

A *virtual appliance* (VA) is defined by Sapuntzakis *et al.* as "like a physical appliance but without the hardware" [39]. Put another way, a VA is a VM designed and configured for a particular task, much like a physical server can be configured with hardware and software with a particular use-case in mind. VAs have the usual benefits of VMs, specifically that they are well isolated, easy to manage and deploy, and are easy to migrate. In order to make these management and deployment tasks fast, however, it is important that the VM disk image size of a VA be as small as possible.

The software appliance concept is an intellectual descendant of other focused small Linux distributions. The Linux Router Project [2] (LRP), for example, took all the software

required to operate a simple Linux-based firewall and router, and packaged it in a distribution small enough to fit on a 1.44MB floppy disk. One of the tools that made the LRP's ambitious goals possible was Busybox[52], a small single-binary application which (via a number of symbolic links) provides simple and small implementations of many typical Linux utilities like `chmod`, `dd`, `grep`, `sed`, `gzip`, `tar`, and many others. While they save a significant amount of space, Busybox implementations of these utilities often do so at the sacrifice of features perceived as less important or rarely used.

In our use-cases, the focus on creating a small, targeted VM image, especially from a full-sized and full-featured physical installation, motivates the development of the *virtual application appliance* (VAA) [46, 47] concept. VAA's are largely distinguished from VAs by their lifetime. A VAA typically starts up, and immediately performs some task (often invoked from a startup script like `/etc/rc.local`), and then shuts down.

The limited scope of a VAA, in particular its focus on automatic execution from beginning-to-end, suggests an avenue for optimization of the size of the image. Because the application must function effectively without interacting with a user, any component of the image that exists exclusively for interaction (e.g., a desktop window manager interface) is redundant. Filesystem usage data that indicates which files are and are not used would immediately allow removal of those components, and a reduction in the size of our VM disk image.

2.2.3 Filesystems as Interfaces to Network Resources

Early network filesystems like the Andrew File System [20] relied on constant network connectivity to retrieve resources from remote systems. As a result, any interruption in connectivity resulted in the entire set of remote resources becoming unavailable, and disrupting the work of users or automated systems using those resources. Such interruptions can be accidental, in the case of a network infrastructure failure, or intentionally planned, as in scheduled maintenance outages. Either way, the effect on those consuming the network resources, and producing data to store on the network, is significant.

Coda [41] is a system designed to enable users in a mobile setting, i.e., where network connectivity is not guaranteed, to have reliable access to remote data. Designed at Carnegie Mellon University by some of the same developers responsible for AFS, Coda adds aggressive whole-file caching to network filesystems, in the hope that server disconnection would not prevent applications from using or storing data in the filesystem. Coda calls this approach *hoarding*, wherein a selection of files is pessimistically gathered from a network-

hosted filesystem, and stored on the local client, chiefly to enable the machine to continue operating during a disconnection from the network. In order to achieve that goal, hoarding grabs as much content as it predicts could be relevant from remote steerage's, limited by local disk storage and network bandwidth. The pessimistic approach is fundamentally different from our goal of a slim appliance, that contains exactly what we know is required, and nothing more.

The Internet Suspend/Resume project (ISR) capitalizes on Coda's focus on making network resources available to mobile computers, but uses it to support mobile **virtual** computers. Since a VM's entire state (from in-memory data structures to simulated hardware devices, to permanent mass storage) is limited to clearly understood regions of a host computer's storage, it is possible to pack up a snapshot of that state, move the entire state to another physical infrastructure, and then resume it. ISR thus allows a virtual workstation to move with a user, making the OS, applications, and data within it available, regardless of where the user is physically located.

2.2.4 Tracing Application Behavior

Orthogonal to the background of virtualization implementation is the topic of tracing application behavior. As discussed above, building a computing infrastructure (whether virtual or physical) which is both efficient and correct requires good data on what resources are used. In the larger body of tracing application behavior, analyzing resource usage is often the responsibility of a profiling tool, such as Gprof, Oprofile, or Dtrace.

Gprof [18], perhaps the best known open-source profiling tool, works by adding instrumentation code at strategic locations as it is compiled by GNU C Compiler (gcc). Oprofile [14], by contrast, implements the profiling code into the kernel. The kernel then records events as executables run on the system, and annotates it with symbol table information included in executables via gcc's "`-g`" option.

Both of these mechanisms suffer from two major problems for our purposes:

1. Neither is designed to record file usage information.

2. Neither approach functions effectively without access to the source code of the programs for which profiling information is desired.

DTrace [10] is a profiling system originally designed to work with Solaris, which has since been ported to Linux [9]. Dtrace introduces a C-like scripting language "D", which allows a developer to write comparatively complex conditions describing what events they

want to profile, including many filesystem events. Outside of Solaris, however, support for DTrace is still early in development. It also requires DTrace-specific kernel modification (via a kernel module), which becomes increasingly complicated, given the simple file-monitoring requirements we have for it.

Scruf [27] is an attempt to move logic into the filesystem. In Scruf, the filesystem can be populated with a series of *trigger scripts*, each of which will be executed in response to filesystem events. As illustrated in Figure 2.3 events filter from the user space through Linux's VFS, which passes events to a modified Ext2 filesystem. In the common case, the modified Ext2 filesystem passes events to a Scruf daemon, which executes relevant scripts based on the action, before finally servicing the request natively.

One of the simplest filesystem extensions provided with Scruf is Scruf-Trace, which could provide us with filesystem usage statistics through its logging of all file opens. For our purposes, trivial modification of the tracing code from Scruf could trace richer use, differentiating between calls to `read`, `write`, `readlink`, and `mknod` (as required for the reasons detailed in Subsection 3.3.4). Scruf, however, relies on modifications to the in-kernel filesystem implementation of the underlying data source. Because our system (Lilliputia) is implemented in FUSE, and takes data from another filesystem source, we can create usage data for any underlying filesystem implementation, and do so regardless of whether the files are local or remote.

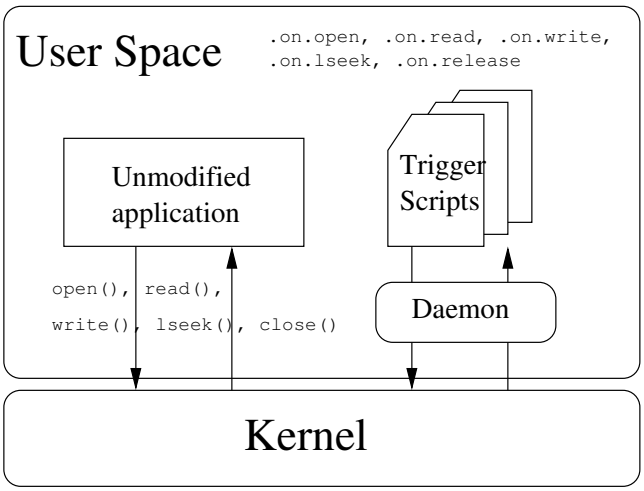

Figure 2.3: Architecture of Scruf

The "ptrace" [32, 33] facility could provide an alternative mechanism for instrumenting filesystem access. Using ptrace, every system call made by an application can be trapped and analyzed before execution. A potential problem with this method is that all system calls

must be trapped, resulting in overhead on all operations, instead of just the required I/O calls. Additionally, `read` and `write` system calls (among others) operate on numeric file descriptor values instead of filenames. Textual filenames are only used at the time the file is opened, meaning that a system to track I/O on individual files would need to remember the file descriptor numbers generated at open time (separately for each process), and then correlate that integer with the number used in `read` and `write` calls. Conversely, under FUSE the file names are provided automatically as part of normal operation.

Another alternative for instrumentation would be the modification of an existing NFS server, and running applications within the mounted NFS filesystem. A modified NFS server could then create a log entry for each I/O operation made by an application. This approach is, in fact, very close to the approach Lilliputia implements with FUSE. The FUSE-based implementation, however, proved simple to construct and debug, especially given our relative experience with the two technologies. Modification of the NFS server would require significant understanding of NFS server internals, whereas FUSE is inherently modularized, with a well-documented API.

2.3 Summary

Existing concepts in virtualization hypervisors and filesystems, and implementations supporting them, are well established and understood. In application to scientific computing problems, these techniques can produce fully-functioning VAs. From that starting point, the following chapters provide details on the tools we developed to trace the filesystem access patterns of these VAs, and produce encapsulated, slimmed clones of them, which we subsequently evaluate for effectiveness and correctness.

Chapter 3

Design and Implementation of Lilliputia

With an understanding of the preceding background and related work in virtualization, filesystems in a virtual machine (VM) context, VM applications and filesystem instrumentation, we can investigate the problem of reducing the size of VM disk images. To appreciate the Lilliputia system, and its process of slimming VMs based on data obtained from our StatFS tracing system, we first explain and motivate our intended goals for a successful slimmed image. In particular, output VM images should be as small as possible while preserving identical behavior of the original machine. We also compare and contrast alternate design decisions (building a VM image package-by-package from the bottom-up) to our approach (taking a full-sized and running VM image, and then removing unnecessary files).

Next we present an obvious and simple approach we call *fresh disk*, which reduces VM size by copying files from a source image to a target.

With the fresh disk approach as a starting point, we refine the technique to include *slimming*, which takes advantage of actual filesystem usage information, based on trace data obtained from StatFS. The architecture of the entire system, which we call *Lilliputia*, is discussed in detail, before we proceed to evaluate it in Chapter 4.

3.1 Goals and Design Decisions

3.1.1 Minimalism in VM Disk Images

For our purposes, the metric of a VM's value is the size of its VM disk image, both before and after compression. The size is important, because it can often limit the feasibility of moving the VM (i.e., via the network, a burned DVD, or USB flash device) as well as the

Transfer Method	Transfer Time
Disk-to-disk Copy	16.9 seconds
Gigabit Ethernet	9.1 seconds
Copy to USB flash	32 minutes
Copy from USB flash	75.5 seconds
Wide-Area-Network	14.6 minutes

Table 3.1: Time to move 1GB of data (a moderate size for a VM) via different methods. In all cases, the average of 5 runs is listed. Disk-to-disk copy was measured from a Seagate 3Gb/s ST3250620NS to a Seagate 3Gb/s ST3750330AS. Gigabit Ethernet was measured between two 2 Gigabit Ethernet machines, with a warm cache, without writing to disk. Copying to and from a USB flash drive was measured with a generic 2GB USB flash drive. Wide-area network (WAN) was measured from a web host in Franklin, Ohio, US, to a web client in Edmonton, Alberta, Canada.

time required to move it (See Table 3.1).

In particular, we measure the size of a *golden master* copy of a VM, which is copied to instantiate an executable VM. Once any VM is deployed, running it will cause the creation of more files and resources, which will increase the size of the VM disk image. In many cases, especially in the context of virtual application appliances (VAAs), growth during execution is not an issue, as the VM will be booted, run, shutdown, and reset to its original slimmed condition, preserving no state other than what is expressly moved out of the machine before shut-down. For example, the standard operation of both the Amazon Elastic Compute Cloud [1] and Eucalyptus [29] cloud-computing infrastructures creates a new copy of a VM, runs it, and then discards the copy entirely.

3.1.2 Completeness in VM Behavior

In addition to VM disk size, the other priority for our slimmed images is completeness, that is, its ability to perform all tasks required of it, and produce output as expected. A slimmed VM is not useful if it cannot perform the desired tasks. A slimmed VM, however, may not need to duplicate every function of the original. For example, a full-size VM may have all the utilities and source code required to recompile some or all of its components. A slimmed version of the same VM might not be expected to do so.

In order to produce a slimmed image that has only the required components, a subset of features and behavior must be defined. The selected set of features defines what activities must take place in a data-gathering trace run. Defining the input of a VM as the trace run activities, we define *completeness* as follows:

For a VM A, let $S(A)$ be the slimmed VM produced by our process. Let I be the set

of all inputs we require our VMs to handle. If $S(A)$ produces the same output as A for all inputs $i \in I$, $S(A)$ is complete. Put simply, a slimmed VM should produce equivalent behavior for all relevant inputs as its source VM.

Other criteria which are usually a concern at VM execution time are memory usage and performance. Creating VMs which perform better with respect to either of these metrics (while important) is outside the goals of Lilliputia. In fact, if our slimmed VMs were to execute faster or with less memory usage than an unslimmed one, it could be evidence that we changed something unintentionally (for example, causing software to skip an important execution stage altogether). Likewise, our implementation should not introduce any *negative* effect on either performance or memory usage.

3.1.3 Top-down vs Bottom-up Construction

A key design decision that guided the implementation of Lilliputia was the "direction" of VM construction. Two alternatives were available, as illustrated in Figure 3.1:

1. **Bottom-up:** First, a minimal operating system (OS) could be installed. Second, each required component, and no more, is added one-at-a-time as needed, until the system is complete.

2. **Top-down:** First, a full-size OS and application suite is installed. Second, all unnecessary components are removed, leaving only the required components.

The *bottom-up* approach is typified by small OS installations, conceptually referred to as *JeOS*, or *Just Enough Operating System* [16]. JeOS variants of Ubuntu [8, 43], OpenSolaris [21], and others are available. Ubuntu JeOS is an Ubuntu Linux derivative designed specifically for the creation of virtual appliances (VAs), consisting of the minimum set of software *packages* (defined below) required to boot the system, perform basic maintenance tasks, and to install additional software. From a minimal base, packages of software can be added one at a time as required. For example, starting from the basic system, an administrator might add print queue management packages and email packages (and anything else they depend on), in order to create a print server capable of printing jobs, and emailing users status updates. It should be noted that Ubuntu JeOS was introduced alongside Ubuntu 7.10, and only existed independently until Ubuntu 8.10, where the concept of a "minimal system" install was integrated directly into the standard "server edition" of Ubuntu's installation CD-ROM images.

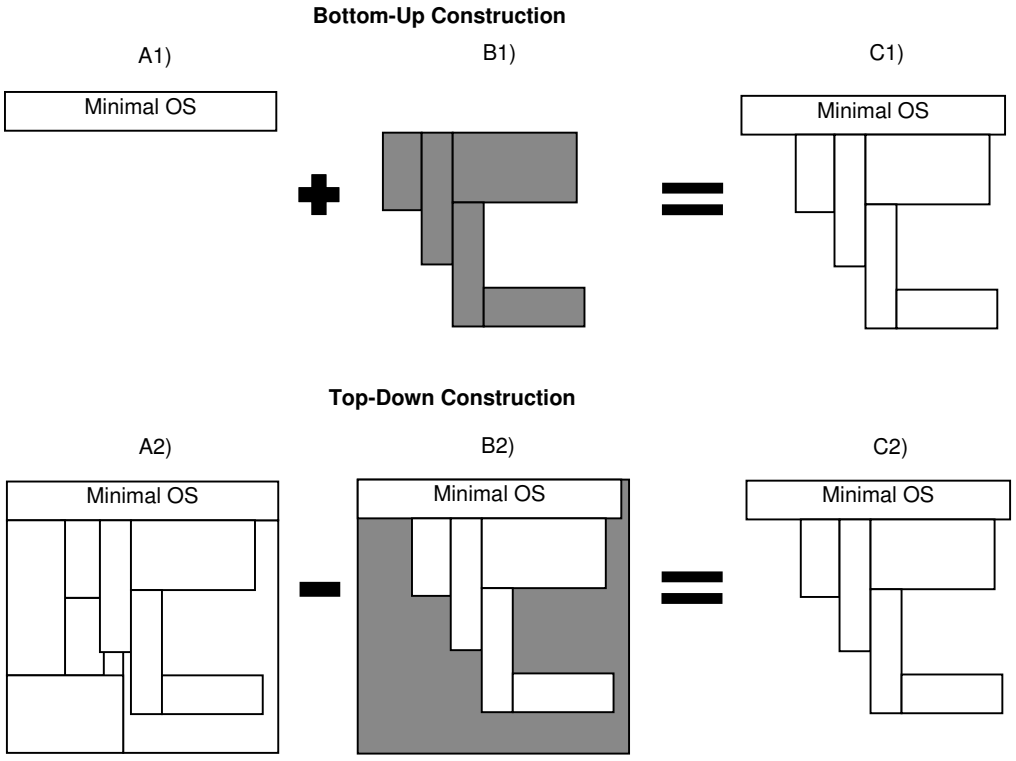

Figure 3.1: Illustration of "top-down" vs "bottom-up" approaches. "Bottom-up", starts with step A1 with the creation of a minimal OS installation. In B1, additional components (in grey) are added one-by-one, producing the final system is step C1. "Top-down" starts with step A2, with a full-sized installation of a working systems. In B2, all components deemed unnecessary (in grey) are removed, again producing the final system in step C2. Note that in practice, C1 and C2 may differ.

In the context of software installation, a *package* refers to a set of interrelated executables, installation scripts, and other files bundled together as a distributable unit. Prior to package management systems like the Red Hat Package Manager (`rpm`) and Debian's Advanced Package Tool (`apt-get`, also used in Ubuntu), the "distributable unit" of most open-source software (even before the term open source was coined) was usually a collection of source code. However, as the dependencies among common software systems became more complicated, an automatic package manager became more important. For example, removing or updating one library package might break executable files from several other packages. Notably, many systems still have some software installed without a package manager, typically because a packaged version is unavailable, but most systems use package managers for most of the software. Nonetheless, one great contribution of package managers is automation, and our encapsulation and slimming system in Lilliputia attempts to automate another tedious task.

A limitation in package-manager-based software installation is that, while installation of software becomes more automated, the list of package dependencies are at least partially documented by a human package maintainer. Hand-generated dependency meta-data presents the following challenges:

1. **Human error:** Like any manually generated meta-data, package dependencies may contain errors. Packages may list dependencies that are not actually required, which can cause the system to include extra files that bloat the size of a disk image. Worse, dependency information may be missing. As a result, an application that is known to work on one installation may fail when installed from scratch in a new location. A missing dependency can cause a failure which is especially difficult to diagnose.

2. **Specific use-cases:** A dependency stating that package X depends on package Y may not be strict, but instead based on general usage. For example, the "network-manager" package in Ubuntu depends on the "ppp" package. Network Manager is responsible for controlling network connections, and PPP is required for dial-up Internet connections. However, if the only use-case an administrator is concerned with is high-speed Ethernet or wireless Internet connection, Network Manager can function perfectly without the PPP package being installed at all. In this case (while the dependency may make sense in general), it may cause unnecessary bloat to the size of a disk image in a specific case.

3. **Package atomicity:** Package managers typically treat a package as an indivisible, atomic unit, which is either completely installed, or completely uninstalled. For example, in Ubuntu's `sed` package (Version 4.2.1-6), the actual `sed` executable is 72 kilobytes (as reported by `du -s`). The rest of the package (which consists entirely of documentation) is 176 kilobytes. In use-cases where the documentation is not required (for example, automatically executing VAA environments), 71% of the package is entirely overhead.

In addition to package-based installation, these challenges often apply equally to manual bottom-up installation of applications (i.e., without relying on a package manager). Redundant software and libraries may be installed, or critical software may be missed during re-installation.

In the sense that bottom-up VM construction requires detailed knowledge of software behavior, and possibly even internal program logic, it is analogous to white-box development. Construction of a small VM disk image without detailed internal knowledge would thus be analogous to a black-box development model, which we call *top-down*.

The top-down approach, unlike bottom-up, takes a fully installed, configured, and working, but potentially unmanageably large installation, and removes elements that are unnecessary just before distribution. For example, a full workstation installation might include everything required to function as a print server, as well as a full suite of other user-oriented applications. Removing all the files related to web browsing, development, and office productivity software would produce a slimmer system, potentially useful as a print server alone. The top-down approach has the advantage of allowing a full-size image to be converted into a slimmed image, instead of creating the slimmed image from scratch.

An analogous example of the bottom-up vs top-down approach is debugging symbols in compiled executables. At compile time, a user executing `gcc` can optionally choose to include descriptive symbol tables inside the executable file with the `-g` flag. Including debugging symbols allows tools like the GNU Debugger (`gdb`), as well as profiling tools discussed earlier, to relate the execution of the binary to the original source code, making it easier to diagnose problems. When it comes time to distribute the binary, it may be reconstructed by `gcc` from scratch, without including these symbols. Omitting the debugging symbols is analogous to the bottom-up approach. Alternatively, the binaries can *always* be constructed including the debugging symbols. In this case, when the size of the executable is an issue, a tool like Unix's `strip` command can remove the symbols (and optionally other data) after the fact. Removing the symbols at this late stage is analogous

to the top-down approach.

Another example of the top-down approach is the Internet Suspend/Resume [40] project, by way of *hoarding* [26] in the Coda [42] filesystem. Using Coda, a selection of files is pessimistically gathered from a network-hosted *full-size* filesystem, and stored on the local client, chiefly to enable the machine to continue operating during a disconnection from the network. In order to achieve that goal, hoarding grabs as much content as it predicts could be relevant from full remote storage, limited by local disk storage and network bandwidth.

The pessimistic approach of copying everything possible however, is fundamentally different from our goal of a slim appliance which contains just what we know is required. It should be noted that the slimming process of Lilliputia, the hoarding solutions in Coda and Internet Suspend/Resume, as well as package-manager systems, all suffer from the same main problem: poor quality or insufficient data and meta-data limits the effectiveness of the solution. Specifically for slimming, incorrect include and exclude lists will either cause the machine to be too large, or incomplete. Our goal then is to easily obtain more complete, high-quality data for a particular user-specific situation. With the limited domain afforded by the context of a VA, we found that we could obtain sufficient data to successfully produce a VA which has everything it needed to function, and still maintain a much smaller disk image size.

3.2 Use-Cases

Although the applications of VM technologies has grown, our work on encapsulation and slimming are mainly targeting the following use-case scenarios:

1. **Physical to Virtual Conversions (P2V).** Many datacentres are converting physical servers into VMs for server consolidation, to support legacy systems when the original hardware needs to be replaced or if the original server build procedure cannot be recreated (e.g., when the procedure has been forgotten or the original employee is no longer available), and to take advantage of cloud computing options. Once a P2V VM is created, systems administrators can also easy replicate the VM instance to maintain server availability or to create mirrors of the server.

   For a stand-alone or self-contained server with only local disks, the P2V process is relatively straightforward: copy the contents of physical disks onto VM disk image in the VM image, then tweak elements like the boot process, device drivers, network configuration, and user accounts.

However, many servers (in our experience) have many network filesystem mounts (e.g., NFS, SMB/CIFS, Andrew File System; we colloquially use NFS as the representative network filesystem). If files on the NFS volumes are used, they must be encapsulated in the VM for completeness. Allowing a VM to NFS-mount a volume is possible (and we have taken this option in some cases), but brings up a large number of security issues and complicates VM migration, especially across administrative domains as in some cloud computing scenarios. In practice, many NFS filesystems may be present, each with many large files and databases. Therefore, including every file which could possibly be used on the entire network would create significantly bloated VM images.

2. **Development VM vs. Production VM.** VMs are useful to encapsulate development environments: special OS versions (either historical or experimental) can be used as the guest OS without affecting the production or host OSes. Similarly, special versions of software libraries or components can be inside the VM without affecting other users or other services. As a development environment, full suites of compilers, tools, libraries, documentation, test harnesses, and header files would be required. Such full-sized environments quickly grow into multiple gigabytes of files, and for production use, the development environment is mostly unnecessary baggage.

   Option 1 would be to reinstall the final software into a production VM (possibly via a version control system) for release. This option is similar to how software is currently deployed in the non-VM world. Often, however, the testing of the software is also done in the development VM. Therefore option 2, slimming the development VM to create the production VM instead of maintaining separate images, has many practical advantages. For example, unforeseen differences (e.g., version of software, paths to software) between VMs can be eliminated since there is only one source VM (i.e., the development VM). Also, if Option 2 is automated (as in the case with our tools), slimming a development VM into production VM is just another step in the build process, after testing.

   For the first weeks of the Trellis Network-Attached-Storage (NAS) Bridge Appliance (TNBA) project in 2006, the developers used Option 1. Predictably, they wasted many hours tracking down why the system worked when testing, but not when they created the production VM. For the later versions of the project, the developers used an early version of the top-down slimming approach. Our contribution is an updated

slimming solution to Option 2.

Of course, the basic idea of tracing file accesses to determine what software is needed (or not) is obvious, and the idea of removing data that is only needed during the development phase goes back at least as far as the aforementioned `strip` command. Our contribution is in finding a practical solution to the completeness (of encapsulation) and minimalism (via slimming) problems and quantifying the benefits with actual VAs. As the implementation details in Section 3.3.4 show, there are also a variety of non-trivial special cases to catch.

3.3 Encapsulation and Slimming

As discussed, the size of a VM disk image is the metric that we are interested in, as it is one of the most important factors in the portability of a VM. The VM disk images(s) store the filesystems that constitute the VM including the guest OS, libraries, applications and data. With default installations of modern OSs ranging from hundreds of megabytes to gigabytes in size, keeping VM disk images within a portable range (e.g., a size conducive to transport via network or portable media) can be difficult and is an ongoing challenge as data and applications are added to the VM over its lifetime. Large VM disk images are slower to copy and provision, as shown in Table 3.1. Importantly, even as device capacity and bandwidth increase in future, disk images will continue to increase in size with increasing demand for features.

Before discussing more complicated techniques of slimming, we will discuss existing methods for reducing VM image size, some of which we will quantify later in Chapter 4.

3.3.1 Existing techniques: Fresh disk and Timestamps

Growable VM disk images formats, like those of VMware and Qemu, accumulate deallocated and fragmented blocks as files are modified and deleted. Therefore, the size of the VM disk images on the host increase over time. VM disk image implementations may provide tools for removing these unused blocks, however, such features are not universal. Some VM disk image implementations do not support reclaiming space at all (e.g., Qemu's "qcow2" format).

A more universal technique we tested, which we call *fresh disk*, was to create a new VM disk image and then clone the filesystem by copying the existing files to the new VM disk image. The idea behind the approach is that creating a fresh copy would not carry with it blocks from deleted files and other overhead that filesystems acquire over time.

The fresh disk approach is not a particularly new or innovative idea. Cloning a disk this way is a common way of eliminating file fragmentation. However, we have found it to be simple and effective in all disk formats, especially when the VM has been in use for some time.

An example cloning script for creating a fresh disk is shown in Listing 3.1. Tools like dump/restore and `tar` can be used, but the standard `cp` program also works. In either case, a few issues must be handled. For example, when duplicating a Linux filesystem, there are two important exclusions: pseudo-filesystems like `/proc`, `/dev`, `/sys` and network-mounted filesystems (as they can be mounted into the cloned VM). Temporary or scratch directories that store transient data may also be excluded depending on the VM's intended use. To exclude these particular directories, the `tar` program supports options for both excluding files (`-X`) and not descending into mounted directories (`--one-file-system`).

```
1  #!/bin/bash
2  # Interactively create a single partition (sdb1) spanning the entire disk.
3  cfdisk /dev/sdb
4
5  # Create filesystem on /dev/sdb1
6  mkfs.ext3 /dev/sdb1
7
8  # Mount filesystem for clone (may need to create mount-point)
9  mount /dev/sdb1 /mnt/clone
10 # directories that should not be copied are written to /tmp/skip
11 echo '/dev/*
12   /sys/*
13   /proc/*' > /tmp/skip
14
15 # clone the root to /mnt/clone excluding certain files
16 tar --one-file-system -cpf - -C / /* -X /tmp/skip \
17   | tar -xpf - -C /mnt/clone
18
19 # install grub on clone by running chroot
20 mount -o bind /dev /mnt/clone/dev
21 mount -o bind /proc /mnt/clone/proc
22 mount -o bind /sys /mnt/clone/sys
23
24 chroot /mnt/clone grub-install
```

Listing 3.1: Interactive Linux script "fresh-disk.sh" to clone `/dev/sda` to `/dev/sdb` for fresh disk

In more detail, our `fresh-disk.sh` script (Listing 3.1) is a tool to perform the fresh-disk cloning operation. `fresh-disk.sh` (where lines starting with "#" are comments) expects a new, unallocated disk has been added to the VM that is assigned the device file `/dev/sdb`. On line 3, the `cfdisk` command is an interactive program to partition the new disk that will be the destination of the clone image of the existing disk (`/dev/sda`). Within `cfdisk` the user must create a partition and set its *bootable* flag. In Linux, partitions are named with an appended number beginning with 1, so the created partition on the clone

disk is named `/dev/sdb1`. Next on line 6, a filesystem is formatted on the partition of type Ext3, and mounted to the mount point `/mnt/clone` on line 9. The `echo` command on lines 11 through 13 creates a list of files not to be cloned (i.e., any files under `/dev/`, `/sys/` or `/proc`). Line 16 (using the `tar` utility) creates an archive from the root (`/`) partition, excluding the files in our list (the `-X` parameter), and passes it to a second `tar` command on line 17, which extracts the archive files into the `/mnt/clone` filesystem.

Once all necessary files are copied to the new VM disk image, the clone disk must have the `grub` bootloader installed. On line 24, `grub` is installed into the new VM disk image's Master Boot Record (MBR) by calling `grub-install` within a `chroot` (change root), using the new disk as the root partition. Special filesystems are remounted inside the `chroot` environment on lines 20 through 22 to allow `grub` to function correctly.

When executed, `fresh-disk.sh` creates a new filesystem on a new VM disk image, with precisely the same files as the original, but with the accumulated overhead of a growable disk removed. Specifically, the size of the VM disk image on the host is approximately the same as the sum of all data-files contained inside the filesystem seen in the virtual guest. For example, consider Tables 4.1 and 4.2. In both tables, the "Host Image" size is notably larger than the "Guest Filesystem" as indicated in the "`/`" row. After running "fresh-disk.sh", the "Post-fresh disk" row indicates that the "Host Image" and Guest Filesystem" sizes are much closer. The same effect also applies as a part of Lilliputia's *slimming* approach (described later).

3.3.2 Timestamp Mark-and-sweep

For the original *slimmed* version of the TNBA in 2006, Michael Closson used a timestamp-based approach to get information on file usage. The access time (`atime`) of all files on the system was set to an arbitrarily old "flag" value, as a "mark" phase. In a following "sweep phase", software is run. Afterward, the filesystem was searched for all files with an `atime` listed that was different from the "flag". In our experience however, modifying and then querying these timestamps proved to be too intrusive. For example, in the NFS case, modifying `atime` values across a network may also have implications for other systems sharing the same files, and administrative controls may actually make it impossible. Changing the access time of a file has implications for filesystem backups (which use `atime`) and depends on filesystem support, which can be especially difficult in shared network filesystems. Consequently, we designed Lilliputia's trace-based approach to be less intrusive than the timestamp-based approach.

3.3.3 Compression

Once the VM disk images are as small as possible, compression using established tools like `gzip` is a natural next step. Table 1.1 shows the effect of compression both before and after Lilliputia was used to slim VM disk images.

3.3.4 StatFS: Filesystem Instrumentation with FUSE

In order to create slimmed VMs that exhibit the same behavior and capabilities as input VMs, we need trace information about what filesystem resources are used. If we include all resources which are used, and ignore everything else, the slimmed VM will exhibit completeness, as discussed in Subsection 3.1.2.

To gather trace logs of file usage within a VM, we developed StatFS to instrument a running Linux system. StatFS is a virtual filesystem implemented via the Linux facility Filesystem in Userspace (FUSE) [44], as previously introduced in Subsection 2.1.5. FUSE allows the binding of a mount-point, for example `/mnt/statfs`, to a userspace process, passing the normal Virtual Filesystem (VFS) calls into userspace, where it is significantly easier to implement new features without in-depth knowledge of kernel internals.

Figure 3.2 illustrates the architecture of FUSE-based systems. A typical FUSE filesystem application uses the `libfuse2` library for utility functions that interact with the FUSE kernel module via the `/dev/fuse` device. The Linux VFS then provides the connection to userspace applications to access the FUSE filesystem. A FUSE filesystem appears and behaves transparently just like a traditional filesystem, making it ideal for tracing without modifying the software we are trying to observe. Implementing filesystems in user space instead of in-kernel also makes changing filesystem logic comparatively easy to implement, develop, and test.

StatFS is based on a simple loopback filesystem included in the examples for the Perl FUSE bindings [34]. The original Perl implementation simply simulates normal VFS calls on virtual files, by performing precisely the same operation on the real files at a different location. In addition to performing the requested operation, StatFS logs the details of `read`, `write`, `readlink`, and `mknod` calls. A simple approach might be to simply catch the `open` calls to determine what resources were used. However (for the reasons detailed later), differentiating between `read` and `write` calls enables us to create a slimmer image. Additionally, `readlink` instrumentation and `mknod` modifications are also required, as detailed later.

The loopback filesystem, with modifications to `read`, `write`, `readlink`, and

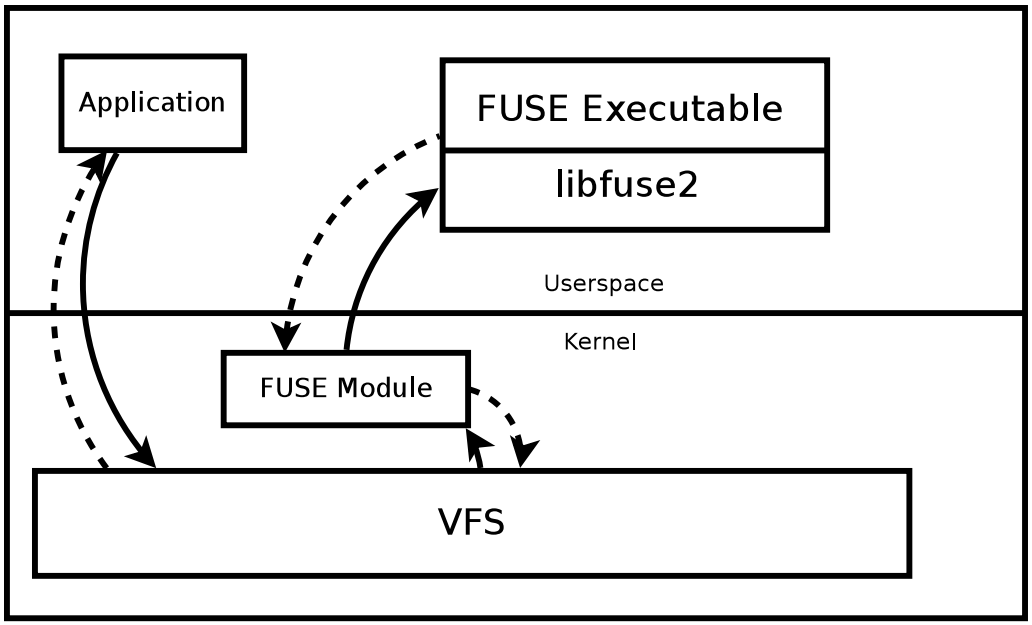

Figure 3.2: Architecture diagram illustrating the interaction between applications, the kernel, the FUSE kernel module, and a FUSE-implementing application (in our case, StatFS). Solid arrows indicate the request from a running application to the FUSE executable, and dashed arrows indicate the response from the FUSE executable to the running program.

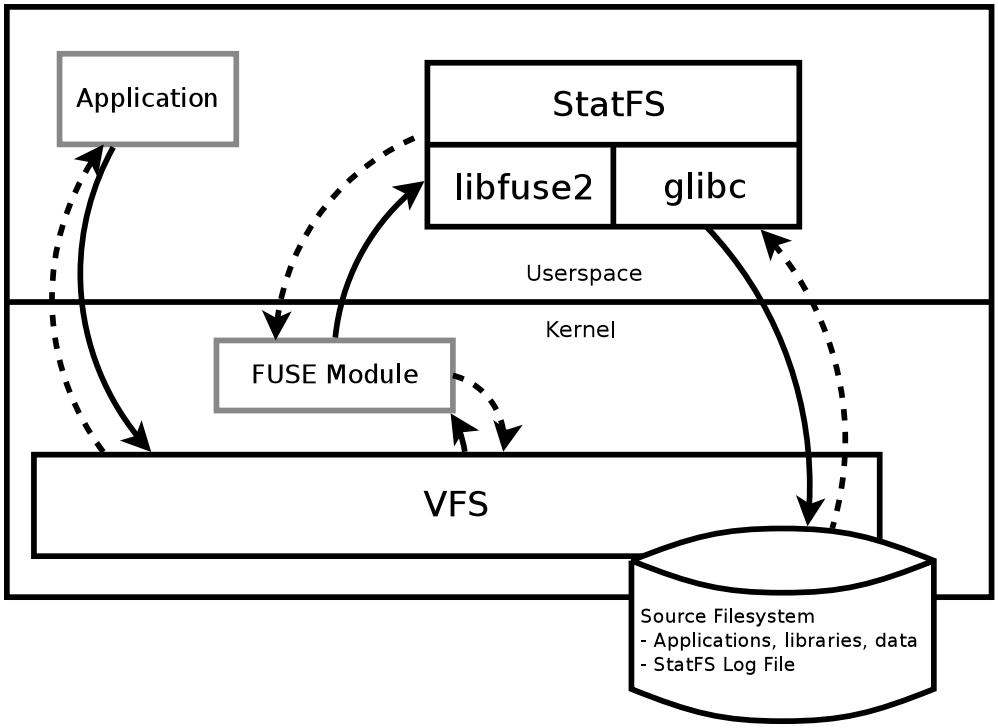

Figure 3.3: Architecture diagram illustrating the implementation of StatFS. Solid arrows indicate the request from a running application through StatFS to the real filesystem, and dashed arrows indicate the response from the real filesystem, through StatFS to the running program.

mknod, (as well as other minor modifications) became StatFS, as illustrated in Figure 3.3.

For example, consider an application reading a file at /cs23d/gafolder/2d3d.pdb. The application will contact the kernel's VFS read function (via glibc), which calls a corresponding function in the Linux FUSE module. The FUSE module calls a userspace function (for example, our statfs_read function) in an executable, typically via the libfuse2 library. Our statfs_read then performs the identical operation on the actual filesystem (again, via glibc) and the kernel's VFS. The result of the call on the actual file is returned in reverse from the VFS, through glibc, statfs_read, libfuse2, the Linux FUSE module, VFS, and finally to the calling application. The only StatFS-specific modification (in this example) is that the statfs_read records the operation to a log file on disk (if the operation was successful).

In detail, the most significant modifications to the loopback filesystem, required to gather desired traces are as follows:

1. **read call:** Instrumentation of the read calls is the most obvious and common situation for most VMs. If the application ever reads a file, that file must be represented later in the slimmed/cloned system if the application is to run equivalently. The requested resource is recorded in the STATFSLOG file, if and only if the requested resource was successfully opened. Instrumentation of the read call is show in Listing 3.2.

```
1  sub statfs_read {
2    my ($file,$bufsize,$off) = @_;
3  # Return an error if the application is trying to read from where we're
4  # mounted (global $mountpoint) ,as this may cause a hall-of-mirrors
5  # effect, in some cases even resulting in an infinite loop.
6    return -ENOENT() if $file =~ /^$mountpoint/;
7
8    return -ENOENT() unless -e ($file = fixup($file));
9    my ($handle) = new IO::File;
10   my ($rv) = -ENOSYS();
11   my ($fsize) = -s $file;
12   return -ENOSYS() unless open($handle,$file);
13   if(seek($handle,$off,SEEK_SET)) {
14     read($handle,$rv,$bufsize);
15   }
16
17 # Log that a file read occured.
18   print STATFSLOG 'read ' . length($rv) . " bytes from $file\n";
19   return $rv;
20 }
```

Listing 3.2: "statfs_read" FUSE call code

2. **write call:** The need to instrument write calls is slightly more subtle. If the application writes to a file, but never actually reads from it, it is **not** necessary to

include the file for completeness. It is, however, important to make sure that the directory the file exists in is recreated, so that the application does not fail if it tries to create that file. It may also be necessary to create an empty file, since the existence or non-existence of the file is an observable difference in a slimmed system. Instead of copying in a complete file that is never read, we could simply create an empty file in its place. In our experiments so far, we have not found a case where recreating empty files was necessary. In either event, the data gathered by StatFS is sufficient to implement the creation of empty files as required. As with the `read` call, the requested resource is recorded in the `STATFSLOG` file, if and only if the requested resource was successfully written. Instrumentation of the `write` call is show in Listing 3.3.

For example, if the output phase of an application creates a file called `result14523.pdb` in the directory `/home/cs23d/output_files/`, we do not need to recreate `result14523.pdb` in order to maintain completeness in the behavior of the slimmed VM. However, we do need to recreate the `/home/cs23d/output_files/`, directory, or the slimmed VM will produce an error when it attempts to write to that file.

```
1  sub statfs_write {
2    my ($file,$buf,$off) = @_;
3  # Return an error if the application is trying to read from where we're
4  # mounted (global $mountpoint) ,as this may cause a hall-of-mirrors
5  # effect, in some cases even resulting in an infinite loop.
6    return -ENOENT() if $file =~ /^$mountpoint/;
7    my ($rv);
8    return -ENOENT() unless -e ($file = fixup($file));
9    my ($fsize) = -s $file;
10   return -ENOSYS() unless open(FILE,'+<',$file);
11   if($rv = seek(FILE,$off,SEEK_SET)) {
12     $rv = print(FILE $buf);
13   }
14   $rv = -ENOSYS() unless $rv;
15   close(FILE);
16
17 # Log that a file write occured.
18   print STATFSLOG 'wrote ' . length($buf) . " bytes to $file\n";
19   return length($buf);
20 }
```

Listing 3.3: "statfs_write" FUSE call code

3. **`readlink` call:** If an application should access a resource via a symbolic link, instead of the file directly, StatFS's output will reflect the I/O operations on the actual files. However, without catching all means by which the application accesses those resources, including symbolic links, the application would not be able to access those

resources without modification. Therefore, we record the usage of a symbolic link as well. Instrumentation of the `readlink` call is show in Listing 3.4.

For example, if an application uses the `jvm` Java Virtual Machine, it may invoke the executable `/usr/bin/java`. However, `/usr/bin/java` (under Debian/Ubuntu systems at least), is a symbolic link to `/etc/alternatives/java`. The Debian/Ubuntu alternatives system [22] allows several packages to be installed which provide the same executable. As a result, `/etc/alternatives/java` can point to any implementation of the Java Virtual Machine, for example, `/usr/lib/jvm/java-6-openjdk/jre/bin/java` from the OpenJDK [31] project. While the previously discussed instrumentation of the `read` call will tell our slimming system that it needs to copy `/usr/lib/jvm/java-6-openjdk/jre/bin/java`, it will not tell us that we need to copy `/etc/alternatives/java` or `/usr/bin/java`. Therefore, instrumentation of the `readlink` call allows us to know which symbolic links to include.

```
1  sub statfs_readlink {
2    my $ln = fixup(shift);
3  # Return an error if the application is trying to read from where we're
4  # mounted (global $mountpoint) ,as this may cause a hall-of-mirrors
5  # effect, in some cases even resulting in an infinite loop.
6    return -ENOENT() if $ln =~ /^$mountpoint/;
7    print STATFSLOG "readlink: $ln\n";
8    return readlink($ln);
9  }
```

Listing 3.4: "statfs_readlink" FUSE call code

4. **`mknod` call:** No instrumentation of the `mknod` call is required, however, we did need to modify the call's behavior. Normally, when an application creates a file, the file's ownership is determined by the user identity of the process that initiated the file creation. With a FUSE filesystem, however, the process that actually creates the file is the userspace FUSE filesystem, and the ownership of the file is accordingly determined by the user identity running the FUSE system. Therefore, while no instrumentation of the `mknod` call is required, it is critical that the file's ownership be changed according to the user identity of the process that initiated the VFS call, so as to ensure that the file is correctly available later on. Modification of the `mknod` call is show in Listing 3.5.

   For example, if an application is run by the user "cs23d" under StatFS, and the application creates a file called `/tmp/intermediate14523.pdb`, it may need to

read the file again later to create output file `/home/cs23d/output_files/result14523.pdb`. However, the file may be created with permissions settings configured such that only the owner of the file may read it. Because the file is created by StatFS, and StatFS is typically run under the administrator or "root" account, `/tmp/intermediate14523.pdb` will be owned by "root", and thus, be unreadable to the user who actually requested that the file be created. Therefore, we change the ownership of the file to "cs23d" immediately after creating the file, and before returning the "successful" result to the application. The same modification is also applied to the "group" permissions on the file.

```
1  sub statfs_mknod {
2      return -ENOSYS() if ! ($can_syscall;
3      my ($file, $modes, $dev) = @_;
4      $file = fixup($file);
5  # Return an error if the application is trying to read from where we're
6  # mounted (global $mountpoint) ,as this may cause a hall-of-mirrors
7  # effect, in some cases even resulting in an infinite loop.
8      return -ENOENT() if $file =~ /^$mountpoint/;
9      $! = 0;
10
11     syscall(&SYS_mknod,$file,$modes,$dev);
12     return $! if $!;
13
14 # Workaround: Because new files are created by the user running this
15 # perl script, other users will see their files owned by someone else.
16 # Whenever we create a file therefore, we chown it to the user that
17 # initiated the FUSE system call.
18 # This approach comes from loggedfs: http://loggedfs.sourceforge.net/
19
20     syscall(
21       &SYS_lchown,
22       $file,
23       fuse_get_context()->{"uid"},
24       fuse_get_context()->{"gid"},
25       $file
26     );
27     return $!;
28 }
```

Listing 3.5: "statfs_mknod" FUSE call code

Finally (and while not currently used by Lilliputia) "open" calls are also instrumented for potential future use.

3.3.5 Tracing Environment

With an implemented trace-gathering filesystem available, it is now possible to actually execute an application within that framework to actually produce useful knowledge about the system. For simple cases, it is possible to "mount" the instrumented filesystem, and then execute the application within it. Two major obstacles present themselves:

1. The application may access files outside the mount-point in the course of its execution

2. Files accessed by the OS before the application is executed will not be noticed.

To gather the most complete picture possible of what resources are used by an application, and the OS that supports it, the entire system (as much as possible) should be run within the trace-gathering framework. We run virtually all system software in our tracing environment via a Linux `chroot` call, wherein the "root" of the observable filesystem is changed for the initial process of the OS (typically `/sbin/init`), and by extension, all child processes. It is possible (via a series of specialized calls [7]) to "escape" from a `chroot` environment, which can limit `chroot`'s effectiveness as a security mechanism. In our case however, we are not using it for security. In the unlikely event that someone wrote into a scientific application the obscure steps required to escape, the only side effect would be accessing resources without recording trace logs.

By directing all filesystem requests inside the `chroot` to our StatFS program, we capture as much of the hosted application's usage patterns as possible. We force almost all software to execute within our `chroot` environment by passing to the Linux kernel at boot time an alternate `init=` option, pointing at a short script, `start.sh`. The `start.sh` script, shown in Listing 3.6, performs a small number of initialization tasks (see below), mounts our tracing filesystem in a new location, performs a few more tasks for compatibility purposes (see below), before executing the system's normal `init` process within a `chroot` into the newly mounted system. The `exec` function in the shell (which is analogous to the POSIX `exec` function) replaces the current process (which, because it is by definition the first process, has a PID or Process ID of 1), with the target process, which obtains the same PID. Making the system init process and its children run in our `chroot` environment limits the amount of information that the normal OS init process has which would indicate that it is running within anything but a normal filesystem.

```
1   #!/bin/sh
2
3   STATFS=/slimming/statfs.pl
4   STATFSMNT=/tmp/statfs/mnt
5   FALLBACK=/bin/bash
6   PATH=/sbin:/usr/sbin:/usr/local/sbin:/bin:/usr/bin:/usr/local/bin
7   STATFSPID=""
8
9   # list of directories to remount under statfs
10  SPECIALDIRS='/dev /proc /sys /tmp';
11
12  # In the event of something catastrophic, start a shell
13  fallback() {
14    echo $1
```

```
15    echo "Starting fallback shell:"
16    exec $FALLBACK
17  }
18
19  if [ $$ -ne "1" ]; then
20    echo 'This script is intended to run as an init process only (PID=1)'
21    echo 'Exiting...'
22    exit
23  fi
24
25  # Make sure the root filesystem is read-write, and set up FUSE
26  mount -o remount,rw / || fallback "Failed to remount / as read-write"
27  modprobe fuse || fallback "Failed to load FUSE module"
28  if [ ! -c /dev/fuse ]; then
29     mknod /dev/fuse c 10 229 || fallback "Failed to create /dev/fuse"
30  fi
31
32  # Start gathering statistics from / by mounting on $STATFSMNT, and recording
        the PID of the statistic tracker fuse application
33  mkdir -p $STATFSMNT || fallback "Failed to create directory $STATFSMNT for
        statfs mountpoint"
34  $STATFS / $STATFSMNT & STATFSPID=$!
35  [ -n "$STATFSPID" ] || fallback "Failed to start \"$STATFS\""
36
37  # Because statfs runs in the background, we'll wait to be sure
38  # that it's up and running before we proceed.
39  while [ ! -e $STATFSMNT/statfs/mpt ]; do
40    echo "Waiting for mountpoint to be up by checking for existence of /statfs/
          mpt"
41    ps $STATFSPID &> /dev/null || fallback "Statfs process died prematurely."
42    sleep 0.5;
43  done
44
45  # Read all the modules in use at this point to catch things statfs may miss
46  for MOD in $(lsmod | awk '{print $1}' | grep -v '^Module$'); do
47    cat $STATFSMNT$(modinfo -n $MOD) > /dev/null
48  done
49
50  # Remount special directories to ensure that they're accessed directly
51  # by future applications, and so we don't see their usage in our log
52  # output anyways.
53  for DIR in $SPECIALDIRS; do
54    mount --rbind $DIR $STATFSMNT$DIR || fallback "Failed to remount $DIR inside
          statfs"
55  done
56
57  # Pass off execution to the system's normal init process
58  exec /usr/sbin/chroot $STATFSMNT /sbin/init || fallback "Failed to chroot and/
        or run init"
```

Listing 3.6: start.sh

The details of allowing a full and ordinary OS within such a `chroot` environment presented a few minor difficulties. One is that the FUSE infrastructure must be prepared, by loading the FUSE device driver into the Linux kernel, and (in systems where it is not done automatically), creating the `/dev/fuse` character device through which user-space FUSE filesystems communicate with the kernel's VFS. Another simple problem is that files may be touched even before the normal `init` process starts. Most modern Linux systems utilize a small "initrd" or "initial-ram-disk" filesystem, which performs the bare-minimum tasks of getting access to the normal root filesystem. These tasks can include obtaining access to

a network filesystem, or simply mounting a local disk. These actions may result in the loading of a small set of drivers into the kernel. To be sure that drivers are seen as "in use" by the trace system, every module loaded in the kernel at this stage is manually read out of the trace-gathering mount-point. Lastly, applications running within the `chroot` may expect to access certain "special" filesystems, notably `/dev`, `/proc` and `/sys`. In modern Linux distributions, these do not represent actual files on any physical disk or network resource. Instead, they are either virtual filesystems which expose an interface for communication with the kernel (for example, getting system information, or setting device driver parameters), or (in the case of `/dev`) in-memory RAM filesystems, managed by other userspace software like `udev` [25]. These special filesystems should not be monitored for usage, and instead are exposed directly within the `chroot` filesystem hierarchy, via calls to the Linux mount option `--rbind`. Mounting with `--rbind` attaches views of these special filesystems to points within the trace gathering system.

The key tasks of the start-up script can be summarized as follows:

1. Load kernel modules
2. Create `/dev/fuse` (if not automatically created)
3. Mount the trace filesystem at a known location
4. Manually read loaded module files from inside the trace filesystem
5. Re-mount (bind) special filesystems to their corresponding locations within the trace filesystem
6. Pass off execution to the OS's normal init process, within a `chroot` call to the trace filesystem's mount-point

With the monitoring system in place, the system can continue its normal boot process. With the OS loaded and ready, in whatever configuration is implemented, the applications can be started, in whatever means are desired. Interactive applications may be started by a human operator. In VAAs, it may mean that applications are automatically executed at the end of a system's normal boot process.

Alternatively (if available), a comprehensive test-suite may be executed. The key to getting a complete trace of application usage is that the system should be stressed in such a way that every resource the administrator expects to be used is touched. In particular, every function of a program should be executed. Given a good test-suite or a simple enough

program, fully stressing the system can be straightforward. In some cases, however, completely stressing every resource of a system may require a significant or even unreasonable amount of effort. In such cases, it may be important to do one of the following:

1. Ensure programs will continue to function acceptably if they are unable to access a file later.

2. Have access to "missing" files via some other method. One implementation of this approach is illustrated in Figure 3.5.

3.3.6 Processing Trace Data

Finally, once the application has been run, and the system shuts itself down, we can actually do something with the trace data we have collected.

First, the raw log produced by the StatFS tracing run is an operation-by-operation log. The log is passed through a perl script, *aggregate.pl* (as shown in Listing 3.7), which builds an "sqlite" database table of file usage statistics. Each row of the table represents an individual file, including the name of the file, the total amount of read and write I/O (in bytes), the number of opens, the number of times the file was used as a symbolic link (if applicable), and the size of the file.

```
1  # Database Handle
2  my $dbh =DBI->connect("dbi:SQLite:dbname=statdb", "", "", { RaiseError => 1,
       AutoCommit => 0);
3  # SQL to check if we've seen a file
4  my $exist = $dbh->prepare("select filename from access_log where filename=?");
5  # SQL to update a file's statistics
6  my $inc = $dbh->prepare("update access_log set read = read + ?, write = write
       + ?, open = open + ?, link = link + ? where filename=?");
7  # SQL to initialize a file's statistics, including its size (as read by perl's
        '-s' function)
8  my $init = $dbh->prepare("insert into access_log (read, write, open, link,
       filename, size) values(0, 0, 0, 0, ?, ?)");
9
10 while (<>) {
11   chomp; s/#.*$//; # Remove comments and newlines
12   if ( /^read ([0-9]*) bytes from (.*)$/ ) {
13     $exist->execute($2);
14     $exist->fetchrow_array() or $init->execute($2, (-s "$2" || 0));
15     $inc->execute($1, 0, 0, 0, $2);
16   } elsif ( /^wrote ([0-9]*) bytes to (.*)$ /) {
17     $exist->execute($2);
18     $exist->fetchrow_array() or $init->execute($2, (-s "$2" || 0));
19     $inc->execute(0, $1, 0, 0, $2);
20   } elsif ( /^opened (.*)$ /) {
21     $exist->execute($1);
22     $exist->fetchrow_array() or $init->execute($2, (-s "$2" || 0));
23     $inc->execute(0, 0, 1, 0, $1);
24   } elsif ( /^readlink: (.*)$ /) {
25     $exist->execute($1);
26     $exist->fetchrow_array() or $init->execute($2, (-s "$2" || 0));
```

```
27        $inc->execute(0, 0, 0, 1, $1);
28      } elsif ( /.+/ ) {
29        print STDERR "Malformed input\n";
30    }
31    $dbh->commit;
```

Listing 3.7: aggregate.pl

Second, we use this database to generate two *include lists*. The first, "sysfiles" (generated by the SQL query in Listing 3.8), is a list of all files which were read, and all symbolic links used, as recorded by the `read` and `readlink` instrumentation from Subsection 3.3.4. The second "sysfiles-all", (generated by the SQL query in Listing 3.9), is a list of all files used, which includes the instrumented `write` calls, in addition to `read` and `readlink`. "sysfiles-all" includes files which may not need to be copied, but do need their parent directories to exist, in case they are written to.

```
select filename from access_log where read > 0 or link > 0;
```

Listing 3.8: SQL query to retrieve files which were read from, as well as all links

```
select filename from access_log;
```

Listing 3.9: SQL query to retrieve all files and links ever used

Our include lists are filtered against an administrator-provided *exclude list*, which is a list of regular expression patterns that will *not* be automatically included. For example, the regular expression "`\.bak$`" files would instruct Lilliputia to avoid copying certain backup files, even if trace data suggests they were necessary. Likewise, the log file `/var/log/syslog` could be excluded with the simple pattern "`^/var/log/syslog$`".

Additionally, a number of files may be selectively included, according to a similar set of regular expression patterns. The *manual include list* in particular allows files to be included even if not observed in the trace data. For example, files used before or after the tracing system is active (i.e., boot-up and shutdown files) can be copied expressly. These typically include:

1. **`/etc/rc0.d`:** Scripts involved in shutting down a Linux system. Included with the regular expression: "`^/etc/rc0\.d/`"
2. **`/etc/rc6.d`:** Scripts involved in rebooting a Linux system. Included with the regular expression: "`^/etc/rc6\.d/`"
3. **`/boot`:** Kernel images, "initial ram-disk" images, and other boot-related files like `grub` modules and configuration. Included with the regular expression: "`^/boot/`"

Shutdown and reboot scripts are necessary to copy because the tracing system may be terminated by the shutdown and reboot processes themselves, therefore it is impossible for the trace system to monitor their use. Likewise, files in `/boot` are generally loaded into memory by the boot loader before the OS kernel even starts, so we have no opportunity to trace their usage. Note that we do not have to manually include the init binary (`/bin/init`), or the startup scripts, because these files are all called inside the `chroot`, after StatFS is active and recording resource usage.

In applying the include list, we also include the targets of any symbolic links matched by the manual include list. For example, (similarly to an example in Subsection 3.3.4) under our Ubuntu installs `/usr/bin/java` is a symbolic link to `/etc/alternatives/java`, which is itself a symbolic link to the actual executable, located at `/usr/lib/jvm/java-6-openjdk/jre/bin/java`. If the manual include-list matches `/usr/bin/java`, both `/etc/alternatives/java` and `/usr/lib/jvm/java-6-openjdk/jre/bin/java` would also included regardless of whether they were present in the trace data.

Finally, we copy the source VM disk to the slimmed target in the actual *slimming* process. Slimming is similar to the fresh disk procedure from Subsection 3.3.1. The main difference is that only resources in the include lists, and no resources from the exclude lists, are actually copied.

3.3.7 Sandboxing Network Trace Runs

A broader problem we worked on solving was the issue of how to create slimmed VM images when network resources are present. Our goal was to create a slimmed VM which could function identically to the "online" version, but with all required resources, and no more, stored locally within the VM disk image. In order to create a "offline", slimmed VM, we need to run our tracing operations, as discussed in Subsection 3.3.5, and create an image that contains files used locally and remotely.

When tracing a development VM that is entirely within one VM, the effects of our tracing operation are, likewise, limited to inside the VM. In the network case, however, applications being observed by our tracing operation may modify data on the network, which could impact other machines. For example, running the development system with tracing may leave data in world-viewable output directories in the production system. In order to protect the rest of the network from the effects of our development VM, we wish to maintain a "sandbox", wherein the development VM can run, make changes to files, and

be successfully traced, without producing any changes to the rest of the network.

To avoid modification of files we want to preserve, we have examined the use of *stacking filesystems* [54]. Such filesystems can combine several different directories, or *branches* on a system into a single, unified view of files. Each branch can have different characteristics, and in particular may permit or deny changes independent of the other branches. For example, consider one directory at the location `/directory1`, containing the files in Listing 3.10, and another directory at the location `/directory2`, containing the files in Listing 3.11.

```
-rw-r--r-- 1 nickurak csusers  23770 2010-08-10 11:57 /directory1/datafileA
-rw-r--r-- 1 nickurak csusers 492871 2010-08-10 11:57 /directory1/datafileB
-rwxr-xr-x 1 nickurak csusers   3452 2010-08-10 11:56 /directory1/executableA
```

Listing 3.10: `/directory1`

```
-rw-r--r-- 1 nickurak csusers 532531 2010-08-10 11:58 /directory2/datafileB
-rw-r--r-- 1 nickurak csusers   1386 2010-08-10 11:57 /directory2/datafileC
-rwxr-xr-x 1 nickurak csusers    363 2010-08-10 11:56 /directory2/executableB
```

Listing 3.11: `/directory2`

If `/directory2` is *stacked* on top of `/directory1`, the unified view, as mounted at `/mnt/union`, would be as in Listing 3.12.

```
-rw-r--r-- 1 nickurak csusers  23770 2010-08-10 11:57 /mnt/union/datafileA
-rw-r--r-- 1 nickurak csusers 532531 2010-08-10 11:58 /mnt/union/datafileB
-rw-r--r-- 1 nickurak csusers   1386 2010-08-10 11:57 /mnt/union/datafileC
-rwxr-xr-x 1 nickurak csusers   3452 2010-08-10 11:56 /mnt/union/executableA
-rwxr-xr-x 1 nickurak csusers    363 2010-08-10 11:56 /mnt/union/executableB
```

Listing 3.12: Unified view of `/directory1` and `/directory2` in `/mnt/union`

Note that the unified view contains files present in either source directory. Where a file exists in both source directories (in this case, `datafileB`) the version from the "top" of the stack is presented in the unified view (distinguishable here by its size).

Further, assume that `/directory1` was configured in the stacked filesystem to be read-only, while `/directory2` was read-write. If an attempt is made to modify `/mnt/union/datafileA`, because `/directory1/datafileA` is read-only, the changes occur in the read-write branch `directory2`, where the changes will still be visible in the unified view. Since the file does not exist in `/directory2`, it will first be copied there, and then modified.

For example, the command in Listing 3.13 will make no changes to `/directory1`, but will cause `/directory2` to be updated as in Listing 3.14, and cause the unified view

to be updated as in Listing 3.15. Note that `datafileA` now appears with its updated size in both the unified view and in `/directory2`, while `directory1` still contains the unaltered `datafileA`.

```
echo "One-more-data-line" >> /mnt/union/datafileA
```

Listing 3.13: Modifying `/mnt/union/datafileA`

```
-rw-r--r-- 1 nickurak csusers  23789 2010-08-10 13:27 /directory2/datafileA
-rw-r--r-- 1 nickurak csusers 532531 2010-08-10 11:58 /directory2/datafileB
-rw-r--r-- 1 nickurak csusers   1386 2010-08-10 11:57 /directory2/datafileC
-rwxr-xr-x 1 nickurak csusers    363 2010-08-10 11:56 /directory2/executableB
```

Listing 3.14: `/directory2` after modifications

```
-rw-r--r-- 1 nickurak csusers  23789 2010-08-10 13:27 /mnt/union/datafileA
-rw-r--r-- 1 nickurak csusers 532531 2010-08-10 11:58 /mnt/union/datafileB
-rw-r--r-- 1 nickurak csusers   1386 2010-08-10 11:57 /mnt/union/datafileC
-rwxr-xr-x 1 nickurak csusers   3452 2010-08-10 11:56 /mnt/union/executableA
-rwxr-xr-x 1 nickurak csusers    363 2010-08-10 11:56 /mnt/union/executableB
```

Listing 3.15: Unified view of `/directory1` and /directory2 in /mnt/union after modifications

The stacking filesystem approach, in our case provided by the FUSE-based "unionfs-fuse" [35] implementation, can thus be used to sandbox the effects of our tracing system on the network. `directory1` above corresponds to network filesystems, which are treated as read-only, while a VM-local writable region is used to store changes. This configuration is outlined in Figure 3.4, which shows the flow of I/O requests. Note that Figure 3.4 is not an architectural diagram since, for example, StatFS and the stacked filesystem live in user-space and the OS (of course) and network live in kernel space.

The command invoked to set up the stacked filesystem in these examples is in Listing 3.16. The "`=ro`" and "`=rw`" options tell whether the branches are read-only or read-write, and the "`-ocow`" enables the "copy-on-write" option.

```
unionfs-fuse -ocow /directory2=rw:/directory1=ro  /mnt/union
```

Listing 3.16: Command line to mount unionfs-fuse for the previous examples

3.3.8 StatFS Pseudo-filesystem Interface

One commonly used function throughout StatFS is `redirect`, shown in Listing 3.17, which alters a requested file path to point at the same file in a different location. For example, if StatFS was configured to provide the contents of `/home/userdata` under the

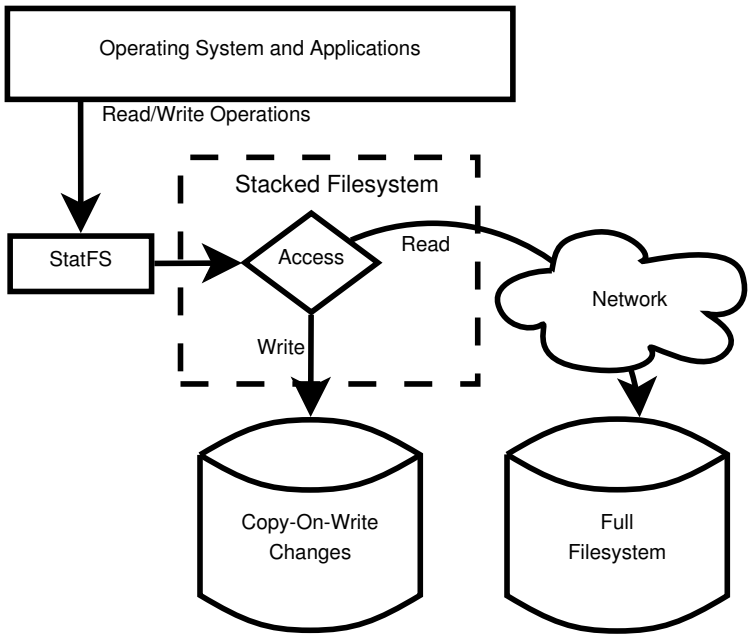

Figure 3.4: Flow diagram of I/O requests with StatFS and stacked filesystems.

mount-point `/mnt/statfs`, `redirect` might translate a request for `/mnt/statfs/directory/file` to `/home/userdata/directory/file`. The code maps requests by pre-pending the requested filename (`directory/file`) with another location, stored in StatFS in `$source_path`. When StatFS is configured to provide the contents of the entire system (`/`), this has no effect. The function is, nonetheless, useful when only a part of a filesystem needs to be monitored.

`redirect` also maps special reserved-word filenames to files in the StatFS installation directory (in this code indicated by `$statfsdir/io`), which can be used to interact with StatFS. Accessing `/statfs/io/cleanup` inside the StatFS mount-point, for example, instructs StatFS to synchronize its log files to disk. Likewise, `/statfs/pid` and `/statfs/mpt` can be used to read the PID and mount point of StatFS, respectively.

```
1   sub redirect {
2     my $path = shift;
3     if ($path =~ /^\/statfs/) {
4       if ($path eq '/statfs/cleanup') {
5         $path = "$statfsdir/io/cleanup";
6         sync_log(*STATFSLOG);
7         return $path;
8       } elsif ($path eq '/statfs/pid') {
9         $path = "$statfsdir/io/pid";
10        open TMP, ">$statfsdir/io/pid";
11        print TMP "$$";
12        close TMP;
13        return $path;
14      } elsif ($path eq '/statfs/mpt') {
15        $path = "$statfsdir/io/mpt";
```

```
16          open TMP, ">$statfsdir/io/mpt";
17    # Save the global $mountpoint into the file
18          print TMP "$mountpoint";
19          close TMP;
20          return $path;
21        }
22      }
23
24      $path = $source_path . $path;
25      # Remove extraneous slashes and dots from the new path
26      do {} while ($path =~ s/\/\.*\//\//g > 0);
27
28      return $path;
29    }
```

Listing 3.17: "redirect" function code

In regular use of the StatFS system, we discovered certain scenarios where it would be beneficial to catch certain requested behaviors of the system, and replace them with modified actions. In Section 3.3.8, for example, requests for the "/statfs/pid" file (which does not exist on the filesystem) are mapped to a file which returns the process id of the StatFS FUSE process.

Another case is the ability to replace binaries on the running system entirely. For example, a straightforward modification to the `statfs_read` call (from Listing 3.2) rewrites requests for the `/bin/umount` and `/bin/kill` programs to a wrapper script inside the StatFS installation directory, `/usr/local/statfs/umount-wrapper.pl` and `/usr/local/statfs/kill-wrapper.pl`, respectively, as seen in Listing 3.18.

```
1       if ($file =~ /^\/bin\/(umount|kill)$/ ) {
2           $file = "$statfsdir" . "/$1-wrapper.pl";
3           print STDERR "Sensitive call, substituting for: $file\n";
4       }
```

Listing 3.18: Modifying file requests for the "umount" and "kill" binaries

```
1       if ($pid == `cat /statfs/pid`) {
2           stat_log("Attempt to kill statfs:")
3           stat_log("Telling statfs to sync our log to disk");
4           system 'touch', "$statfsmnt" . '/statfs/cleanup';
5           exit 0;
6       }
```

Listing 3.19: Catching and responding to request to kill StatFS via "/bin/kill" in the "/usr/local/statfs/kill-wrapper" script

Rewriting requests to unmount the filesystem or kill processes proved useful later in experiments, when it was deemed useful to synchronize data (for example, our StatFS log file) when the system attempted to kill the StatFS program or unmount its filesystem. In this case, the `/bin/kill` program is replaced with a wrapper, partly represented in Listing 3.19, which uses the earlier discussed "cleanup" function of StatFS, and subsequently

ignore the kill call. A similar wrapper call can be used to ignore a request to unmount the FUSE StatFS filesystem. These particular choices for wrapped calls are mostly illustrative. This approach only catches requests to kill or unmount the StatFS filesystem via the standard binary executables at `/bin/kill` and `/bin/umount`. Notably, it does not catch attempts to kill or unmount StatFS via syscalls. In our experience, this limitation is not significant, as the typical approach to killing processes or unmounting filesystems is for a script to call the binary in `/bin`, and not execute the syscall directly.

3.3.9 On-Demand Resources for Incomplete Virtual Machines

Lilliputia, as implemented, is able to construct a VM that contains everything indicated by a complete trace run. In some scenarios, however, it may be necessary to knowingly create an *incomplete* slimmed VM, which is missing components. For example:

1. It may be impossible or infeasible to produce trace runs with complete coverage. If so, files which may eventually be needed in production use may never be detected in a tracing run.

2. A complete image, even after slimming, may still be too large for some use-cases. Lilliputia's output database indicates the total amount of I/O used for each file, and can therefore suggest files which, although used, are only used rarely.

In either case, files which may still be required eventually, but cannot be included in the slimmed image, may still be made available in the network. Stacked filesystems (as described in Subsection 3.3.7) can use these network resources to fill in the gaps in an incomplete VM, as illustrated in Figure 3.5. If a file is required by an application, but not present in the slimmed image, the filesystem can transparently fail-over to retrieving the file over the network, with the usual speed penalties of retrieving those missing files.

3.4 Summary

In order to create VMs that can be easily and quickly provisioned, deployed, and migrated, it is important to keep the VM disk images as small as possible. Minimizing the size of these images from large development installations dictates a top-down approach, which requires an ability to detect which resources within the VMs disks are important to their functionality. StatFS, as implemented and documented above, gives us the ability to generate appropriate trace data, which we can then use to clone the important parts of an input VM, producing a slimmed VM in the complete Lilliputia system.

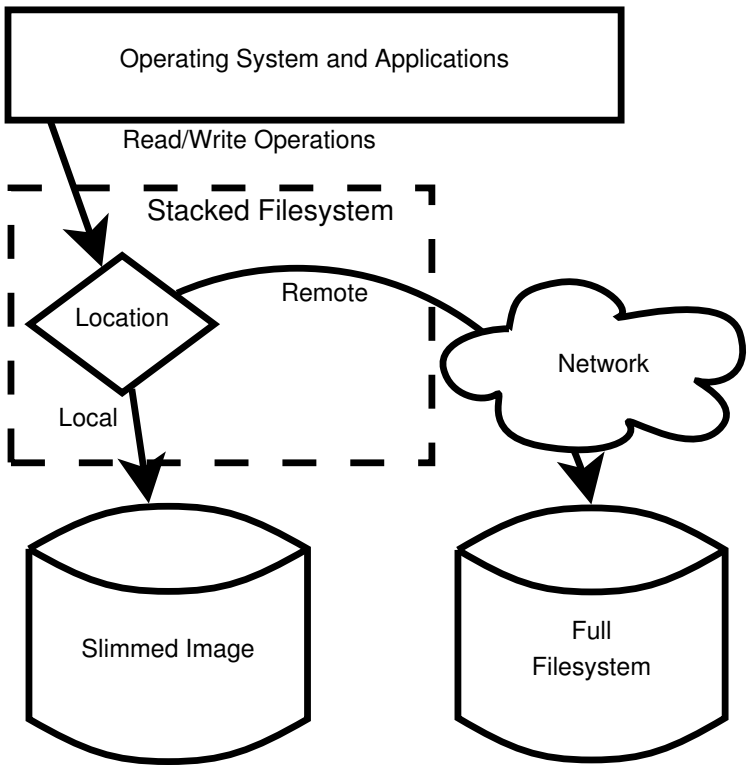

Figure 3.5: Flow diagram of I/O requests with a slimmed image and network fail-over.

In order to establish the effectiveness and correctness of Lilliputia, Chapter 4 details our empirical evaluation methods and results.

Chapter 4

Empirical Evaluation

As discussed in Section 3.1, the goal the slimming approach in Lilliputia is to *minimize* the size of a virtual machine (VM) disk image, without sacrificing the *completeness* of its functionality. In this chapter, we present a case study of our experiences applying Lilliputia to two different VM and virtual appliance (VA) systems, the Trellis Network-Attached-Storage (NAS) Bridge Appliance (TNBA), and the Chemical Shift to 3D Structure (CS23D) server.

The main results of our evaluation show a significant reduction in VM image size, and verify that VM functionality remains intact. In particular, the size of our VM disk images (after compression) was reduced by 95% in the case of the TNBA (from 591 to 27 megabytes), 94% in the case of the CS23D appliance (from 32 GB to 2GB), as mentioned earlier in Table 1.1, and detailed in Tables 4.1 and 4.2. These slimmed images should be markedly faster and easier to move across the network, or via physical media (i.e., USB flash drives or writable DVD).

All experiments were performed on a 2.4GHz Intel Core 2 Quad CPU machine with 8 GB of RAM and a Seagate 3GB/s ST3750330AS, running Linux 2.6.30 on Fedora 11 (within our department named `codesa.cs.ualberta.ca`). However, with the exception of the timing details in Section 4.5, all the VM image size results in this chapter are independent of host system performance.

4.1 Case Study Appliances

To evaluate the effectiveness and correctness of Lilliputia's slimming in diverse settings, we applied it to two VA environments we discussed earlier. First, the TNBA, a VA for making it easy to access network resources, was developed by our research group in 2006, and was well understood by our group. Second, CS23D, a web server for protein structure predic-

tion, was provided to us by a collaborating research group. The TNBA and CS23D VM disk images were also constructed using different approaches, increasing confidence that the slimming procedure of Lilliputia is effective and correct in a wide range of applications.

4.1.1 TrellisNBA

The TNBA is a VA designed to easily connect to remote filesystems across different administrative domains. The basic architecture and flow of control of the TNBA is illustrated in Figure 4.1. The appliance presents a *Server Message Block/Common Internet Filesystem* (SMB/CIFS, the protocol for network filesystems typically used in Windows networks) interface for accessing files available under other SMB/CIFS domains, as well as Secure Shell (SSH) file systems. The TNBA uses the Trellis Filesystem to present a unified POSIX-like front-end to cached copies of remote files. The files can be accessed via a SMB/CIFS back-end to make access to these resources simple under any system with support for both virtualization and normal Windows file-sharing protocols. It also provides a web-based configuration and management interface.

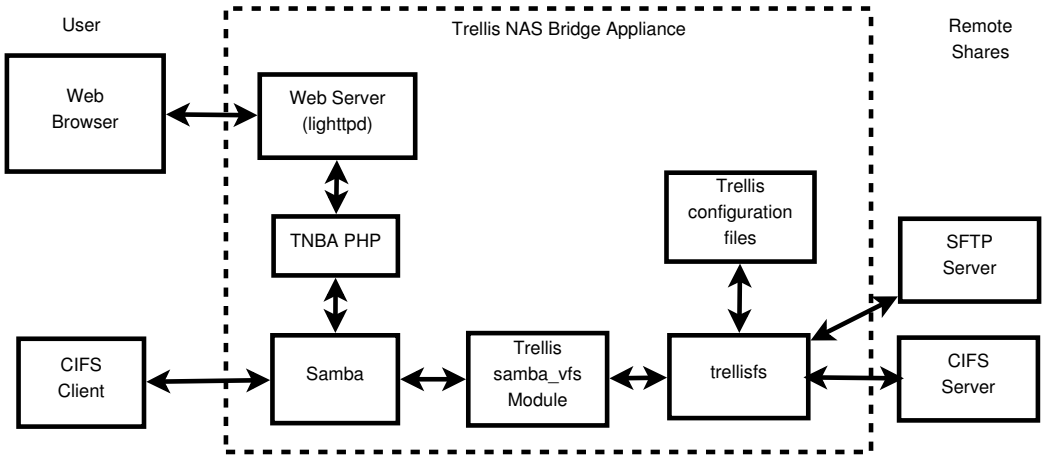

Figure 4.1: Flow diagram of the Trellis Network-Attached-Storage (NAS) Bridge Appliance

The deployment of the TNBA used in our experiments was built from the bottom up (as discussed in Section 3.1.3). The initial operating system was a *JeOS* [8] installation of Ubuntu Linux 8.04. Ubuntu JeOS contains only the bare minimum packages required to boot the operating system and provide basic support for hardware (or in this case, virtualized or emulated hardware). A baseline JeOS installation is under 500 MB installed, and approximately 100 MB compressed. On top of that, hand-picked packages are added one at a time to support the compilation and installation of the TNBA. Our additions include a web

server (lighttpd [24]), a file-sharing service (Samba), and supporting libraries and utilities. The full list of packages added to support installation and execution of the TrellisNBA was: php5-cgi, lighttpd, libc6-dev, gcc, make, libgamin-dev, libdb4.5-dev, libssl-dev, zlib1g-dev, wget, automake, autoconf, libtool, tcsh, libsmbclient, libdumbnet-dev, libproc-dev, libicu-dev, apache2-utils, openssh-client, openssh-server, fuse-utils, and sshfs.

The TNBA was originally developed within our research group as a modification of the FreeNAS [13] appliance. As such, we had a significant amount of knowledge of its components and behavior. Accordingly, the TNBA was an ideal environment for development, debugging, and fine-tuning of our Lilliputia system.

Because the construction of the TNBA was a bottom-up process, it is normally an ideally situation for VA creation:

1. The TNBA installation is relatively simple, with a small number of components
2. The TNBA installation was built by a single administrator, with in-depth knowledge of what is required and what isn't.
3. All development and debugging work was performed before installing and configuring the system

As a result, the TNBA is naturally a small appliance. Conversely, its small size means that the TNBA is a challenging case for Lilliputia – the bottom-up approach implies that there should be fewer opportunities to remove unrequired resources, since a large number of extraneous packages was never installed in the first place.

4.1.2 CS23D

CS23D [53], by contrast, is a significantly larger and more complicated service and application suite. CS23D is a protein structure predictor, developed by a number of bioinformatics researchers. The VM we worked with required over 2 months to configure all the components to work together, with input from a number of developers. Notably, creating the initial CS23D VM differed from the TNBA case in that it was *not* a bottom-up process: many development packages and multiple versions of libraries and utilities were included over its lifetime before it was completed.

Among the applications included in CS23D are Proteus2, Preditor, Rosetta, SFAssembler, GAfolder, and PatterNOE. Many of these applications have their own collection of large protein databases. These applications and the CS23D framework include interpreted

and compiled code in many languages (C, Java, Perl, Python), meaning that large and diverse development environments are required to work with and deploy the system as a whole.

For its user interface, CS23D uses a Web-browser based front-end. Users supply an input "chemical shift" file in `.str` format to an HTTP CGI application, which stores the input file, creates a job in a database, and processes the input to create an output file in `.pdb` format, which is finally emailed to the user who requested it.

The various components were also produced by a large group of developers external to our group over several years, each with different goals and environment preferences. In the over 2 months developers required to get everything to interact correctly, many different tools, libraries, and even versions of software were installed and configured separately. Since our group did not have experience with the wide range of distinct components, or indeed the completed VM provided to us, CS23D was an ideal "black box" for evaluating how applicable Lilliputia would be in general.

The large suite of applications also means a significantly varied set of file access behaviors. Applications use traditional MySQL databases, as well as direct access to files of many different sizes, making it a challenging test for the correctness of Lilliputia.

Finally, because of the complexity of CS23D, it is often infeasible to hand-pick which parts of the installed base contribute to the execution of the appliance, as compared to those parts which are simply required for its compilation and configuration before deployment. As such, the image of CS23D we began work with was a complete development environment with many tools and utilities, as installed under a full installation of Fedora 11 Linux.

After including all the applications, databases, and supporting libraries, tools, and development environments, CS23D requires a 42 gigabyte VM disk image.

4.2 Evaluation Criteria

The criteria by which we evaluated our slimmed VMs was the size of their VM disk image, as measured while shut down, as described in Subsections 3.1.1 and 3.1.2, since that metric limits our ability to move and provision the VMs. As discussed in those subsections, other performance metrics like execution time and memory footprint are important, however, they are not the focus of this evaluation. In particular, we do not attempt to reduce any virtualization-related overheads, as the filesystem calls in our slimmed VMs will be identical to those in the original unslimmed VMs.

In comparing VM disk images in this chapter, we take care to compare our final post-slimming results to the results of *fresh-disk* version of their original VM disk images. The fresh-disk versions were used because the fresh-disk approach is an obvious technique, and it was our desire to evaluate Lilliputia's slimming process independently.

It should be noted that in all cases, "swap" devices for all VMs were maintained as separate VM disk images, and since they do not store any actual content between VM invocations, they can be deleted and recreated as a blank, unused disk image, which consumes virtually no space. As such, the swap devices are not included in totals before or after slimming.

4.3 Experimental Methodology

The above applications were tested with Lilliputia by means of the following methodology:

1. **Installation:**

   In the case of the TNBA, we build the input VM disk image bottom-up. The base Ubuntu JeOS install was performed on a blank VM disk. Supporting packages were installed via `apt-get` as discussed above, including a compiler, web server environment, and libraries. Finally, the TNBA (including Samba components) was compiled, installed, and configured. Before slimming, the TNBA image was 591 MB uncompressed, and 242 MB compressed, as in Table 1.1.

   For CS23D, a complete Fedora 11-based VM image was provided to us by an external developer. This image included a number of applications and libraries installed with the `yum` package manager, and a large suite of manually compiled and installed bioinformatics software. Before slimming, the CS23D image was 32 GB uncompressed, and 9.3 GB compressed, as in Table 1.1.

   In both cases, the VM images were confirmed to be working correctly, as per the verification section discussed later.

2. **System booting:** The installed systems were booted under StatFS, via the kernel `init` option, discussed in Section 3.3.5.

3. **Tracing:** The systems were run through their normal operating process, to create trace information for the application.

   For the TNBA, trace execution involved using every feature in the system, through both its web front-end and its SMB/CIFS interface. The web front-end was used to

configure user accounts and groups (creating, reading, updating, and deleting them), configure remote Secure File Transfer Protocol (SFTP) and SMB/CIFS shares (creating, reading, updating, and deleting them), use the TNBA's diagnostic tools (ping, traceroute, log file access), read TNBA documentation, and eventually, shut down the VA. Before shutting down the VA, the configured shares were also accessed remotely via TNBA's CIFS interface (again, creating, reading, updating, and deleting files).

For CS23D, the process was much simpler: Because CS23D operates largely non-interactively, we manually provided an input file to CS23D, and retrieved the output file when it was finished. The system was then shut-down via the normal shut-down facility, `/usr/bin/halt`.

4. **Slimming and Cloning:** After tracing was complete, the data processing and actual cloning processes from Section 3.3.6 were applied.

   First, the trace log from StatFS was retrieved, and copied into a copy of the original source VM, which had the StatFS system, but had not had tracing steps run on it, to make sure that the final output VM did not show evidence of the trace system having been run on it. Additionally, a new, unformatted and unpartitioned VM disk image is added to the VM. `aggregate.pl` is run on the trace log, which generates the database of file resource use. Finally, this database is used to select which files to copy from the original, unaltered system, to the blank disk, which is configured to boot with `grub`. After this step, the new VM disk image is a complete, slimmed, bootable clone of the original input disk.

5. **Compression:** The final output disk images were compressed using the `gzip` utility. For completeness, the same compression process was applied to the unslimmed VM disk image as well, as in Table 1.1.

6. **Validation:** The slimmed disk image was executed and checked for completeness, as described below in Section 4.4.

4.4 Validation

In order to confirm the completeness of the final slimmed images from both the TNBA and CS23D, several inputs were used with both non-slimmed and slimmed VMs to check if they produced the same output or behavior for the same input. The main validation steps were

Stage	Full Size		Slimmed	
	Guest Filesystem	Host Image	Guest Filesystem	Host Image
/	581	767	49	76
/usr	272		27	
bin	23		7.2	
lib	118		12	
local	47		7.1	
share	71		0.094	
(other)	11		0.81	
/boot	14		12	
/home	110		0.0039	
/var	137		0.32	
/lib	35		5.6	
/etc	6.3		1.1	
/bin	3.8		2.2	
/sbin	3.5		1.2	
(other)	2.2		0.062	
Post-fresh disk	581	591	49	54
Compressed	241	242	26	27

Table 4.1: Trellis Network-Attached-Storage (NAS) Bridge Appliance Slimming. All sizes are in MB, as reported by `du -s`, and compression is via gzip

effectively the same actions as the tracing steps from Section 4.3: if the tracing steps were complete enough, and the behavior of the slimmed VMs matched that of the unslimmed VMs for all inputs, the slimmed VMs could be considered valid.

As stated in Section 1.4, our approach to encapsulation and minimalism for VMs depends on proper trace coverage. In scenarios where tracing data is incomplete, the slimmed VM will be in one of three states:

1. Manual include lists will be required to make the slimmed VM complete
2. On-demand access to missing resources must be configured, for example, over a network-filesystem, as discussed later in 4.5.2
3. The VM will be incomplete, and produce incorrect behavior

Note that for this phase, the tracing system was inactive, since the tracing data had already been collected and processed.

4.4.1 TrellisNBA

Since the TNBA tracing steps were straightforward, the exact same steps were taken in the unslimmed and slimmed TNBA VMs. Again, this involved using the TNBA's web front

	Full Size		Slimmed	
Stage	Guest Filesystem	Host Image	Guest Filesystem	Host Image
/	30666	42823	8234	9800
/cs23d	24750		7972	
/usr	3331		121	
share	1578		4.1	
lib	1393		111	
bin	177		1.6	
include	70		0	
libexec	50		0.39	
sbin	28		4.0	
local	0.54		0.055	
(other)	0.48		0.0039	
/var	1642		42	
/home	509		0.66	
/opt	178		66	
/etc	92		3.1	
/lib	84		15	
/root	18		0.031	
/tmp	17		0.0039	
/sbin	14		4.2	
/boot	8.3		8.3	
/bin	6.8		3.8	
(other)	16		0.0039	
Post-fresh disk	30666	31876	8234	9366
Compressed	9226	9305	2025	2054

Table 4.2: CS23D Slimming. All sizes are in MB, as reported by `du -s`, and compression is via gzip

end to configure user accounts and groups (creating, reading, updating, and deleting them), configure remote SFTP and SMB/CIFS shares (creating, reading, updating, and deleting them), use the TNBA's diagnostic tools (ping, traceroute, log file access), read TNBA documentation, then accessing configured shares remotely via TNBA's CIFS interface (again, creating, reading, updating, and deleting files), and finally, shutting down the VA.

Identical behavior was noted in the both unslimmed and slimmed TNBA VMs.

4.4.2 CS23D

Similarly to the TNBA, our slimmed and unslimmed CS23D VMs were executed with the same inputs as in the tracing stage, and their outputs compared.

Validation of correctness for CS23D is complicated by the use of random number seeds in a number of the fundamental algorithms of its constituent applications. The usual approach of manually selecting seeds is also less feasible, due to the large number of distinct pieces of software, and the breadth of programming languages, tools, and expertise required to effectively manage it. Therefore two approaches were taken to confirm correctness. First, execution of the process was monitored via log files to observe the same programs being executed in the same way on the same input. Second, the final outputs were examined by a CS23D developer (external to our group) to confirm that an acceptable quality was produced by the tool chain. Given the design of CS23D, wherein each program contributes to making a better output file, we conclude that it is unlikely a removal of normally included functionality would improve the final result.

4.5 Discussion of Results

As shown in detail in Tables 4.1 and 4.2, the size of our VM disk images (after compression) was reduced by 95% in the case of the TNBA (from 591 to 27 megabytes), and 94% in the case of the CS23D appliance (from 32 GB to 2GB), as originally illustrated in Table 1.1. While the overall space-saving results are similar, it is informative to consider the distinctions in *where* space was saved in the TNBA and CS23D cases.

4.5.1 TrellisNBA

The results of applying the Lilliputia slimming process to the TNBA are shown in Table 4.1, including a breakdown of which sub-folders of the appliance contributed to the size of the image, both before and after slimming. As mentioned, the bottom-up construction of the TNBA represents a challenging scenario for Lilliputia. Nonetheless, the results of our

experiments show it to still be successful. This results from a combination of removing parts of normally atomic packages, removal of packages irrelevant to our use case, and potentially erroneous dependencies, as discussed in Subsection 3.1.3

The main conclusion and observation is that Lilliputia reduced the TNBA from a 591 MB virtual image, down to a 54 MB uncompressed image, for a savings of over 91%. For actual distribution, the files would typically be compressed down to 27 MB through use of `gzip`.

Some major sources of unnecessary files indicated by Table 4.1 are a large number of libraries under `/usr/lib`, where 90% of content was removed, and the almost complete removal of cache and log files under `/var`, which were either not accessed when the appliance was used, or were only used in a write-only capacity, and were therefore not required to maintain normal application function. Another major location of slimmed-away resources was `/usr/share`, which was also almost completely removed. `/usr/share`, as defined by the UNIX Filesystem Hierarchy Standard, includes architecture-independent data files, such as textual data or images, as well as locale and timezone information. It also typically includes a large amount of documentation. Because virtually none of these resources were relevant to the successful operation of the TNBA (documentation is largely for the benefit of a user logged into the system directly, and the only locale and timezone information relevant was that which the system was actually configured to use), it should not come as a surprise that they presented a substantial opportunity for image size savings.

The other major opportunities for savings were in files not used to operate the appliance, but simply to deploy it. These include source code, statically-linked libraries in `/usr/lib` and `/lib`, compilers, and all the supporting development tools required to operate even a simple Linux system. Once the applications are deployed, however, none of these are required again. `/home` was also almost completely removed, as it was mostly used simply for initial installation of the TNBA (for example, the TNBA source code).

4.5.2 CS23D

Likewise, results of applying Lilliputia to CS23D are shown in Table 4.2, again including a breakdown of the major contributors to image size.

A notable addition to Table 4.2 table as compared to Table 4.1 is that CS23D stores many of its application and data files in a `/cs23d` directory, which was reduced in size by 68%. Similarly to the TNBA case, large savings were also observed in `/usr/lib` (92% savings) and `/usr/share` (over 99% savings).

An important difference in the way CS23D operates is that it is meant to handle a particular input file, provided as a query, and return a result. While in the TNBA we were able to exhaustively enumerate all the possible configurations of feature usage, it is impossible to assemble all possible input files that could ever be supplied to CS23D at any time in the future. Doing so would require an infeasible amount of time to execute each query independently. Worse, validation of each of the possible results would require an incredible amount of manual effort. Due to time constraints on availability of researchers in the domain, we were in fact only able to confirm the correctness of a single input file.

To ensure correctness of the output VM, we followed the flow of a single input file through all phases of CS23D's operation, and (as discussed in Subsection 4.4.2) confirmed that the result met the same standard of quality as the original system. This confirmed that the slimmed appliance was able to successfully process that input file, but did not necessarily imply its usefulness to other queries. When faced with such a limited range of inputs, two approaches we considered were as follows:

1. **Randomly generate more inputs:** Where possible, "fuzzing" [28, 30] might be used to generate more candidate inputs for the VM tracing process. Fuzzing is an approach typically used in software testing and security, wherein a number of randomized inputs are generated and fed into a system. The results of these inputs can be observed for potential faults in the implementation of a system, or even potential security vulnerabilities. Two constraints made this difficult in the CS23D case. First, randomly generating inputs might produce invalid or highly unusual inputs, which would never be encountered in real-world usage. Such inputs could cause the VM image size to bloat beyond the size actually required. Second, we did not have the expertise to tell how to successfully generate or evaluate CS23D input files for correctness.

2. **Manual include lists:** By observing the execution of CS23D, we were able to generate a manual include list that should broaden the applicability of our slimmed VM, and avoid under-reporting the size of an appliance required to handle general inputs. To do so, we manually added files to our include-list that would be necessary for other inputs. Our include list was created by examining the resources used in the process of solving the one input, and broadening the list of included files. In particular, if the input was found to use even one small part of a large database, the entire database was added to our include-list. Thus, erring on the side of caution, it is possible we actually included more database files than would likely ever be used.

Alternatively, in the event a comprehensive test-suite is not available, and no manual intervention can be applied, a similar stacked-filesystem approach to that illustrated earlier can be used, as in Figure 3.5. As discussed in Subsection 3.3.9, stacked filesystems can transparently layer in files that are available over the network, even if they are not available in the slimmed image. Stacking filesystems can allow an administrator to leave out files in the slimmed image that would only be used occasionally (either through manual exclude lists, or by limited tracing), but still have them available where necessary.

In general, an automated and comprehensive test-suite is always highly desirable, and is particularly beneficial when deriving file usage. Had one been available at the time, a complete test-suite would have yielded a more specific set of file resources needed for execution of the appliance than manual inclusion would provide. Having verified the correctness (but not the optimality) of our slimmed appliance, we can consider how well the slimming process performed. As in the case of the TNBA above, a significant portion of the files removed from the appliance were development tools, compilers, libraries, and source code. The prevalence of such files in CS23D was a result of the fact that CS23D utilizes a much broader range of development tools, for a comparatively large number of distinct components, as built by a large number of independent research groups, each with different goals and development styles.

Nonetheless, a relatively large amount of the original application remains, as the appliance only changed from 31.1 GB to 9.4 GB, a savings of only 70%, which while significant, is not nearly as impressive as in the TNBA case. By examining the slimming process without the discussed manual include-list above, we found that the slimmed image would have been just over 400 MB. The difference between 9.4 GB and 400 MB means that the vast majority of the slimmed image's heft is a result of the manually included database files, which we were unable at this time to avoid cloning. However, the format of these files was large plain-text databases of protein information, meaning that the effect of compression in the final step was dramatic. As a result, the final savings overall from 31.1 gigabytes to 2.1 gigabytes is over 93%, which is comparable to the 95% overall savings observed in the TNBA case.

4.5.3 Performance Results

Some discussion is warranted regarding the effect of Lilliputia, and StatFS instrumentation method on the performance of a system. Measuring time in the TNBA case was not particular useful, as the application was primarily interactive, meaning the user response time

vastly overwhelmed any difference in application performance caused by the trace operation. CS23D, by contrast, is an entirely non-interactive application. A single CS23D trace run within an instrumented, local filesystem, required 113 minutes to execute. The same invocation without any instrumentation required 109 minutes, an overhead of 3.7%. At this stage, we have focused on correctness more than performance of the system, as the instrumented runs are not intended to be run at production time. However, these results suggest that, at least for CPU-bound scientific applications, instrumentation has little overhead, and may be feasible for use in production systems as well. Including StatFS at production time would have the advantage of producing live traces of an application's resource usage, which could in future work be used for additional performance tuning.

Finally, the process of actually producing a cloned image from the traces gathered also takes a small amount of time. In the relatively complex case of CS23D, the plain-text trace file came to 10 megabytes. Producing aggregate per-file statistics required 13.2 seconds (an average of 5 runs) to build a 357 KB statistics database. Producing the list of files to include from this database alone was virtually instantaneous. However, the rich regular-expression rules for hand-coded include/exclude lists requires a full scan of the source system's filesystem hierarchy for matching names. Comparing each of these to all the hand-written rules is therefore more taxing. In the case of CS23D searching the filesystem for matching files took approximately 5 minutes, producing a 187 KB list of files to copy, and a 231 KB list of all files read or written (required as discussed in Section 3.3.4 under instrumentation of the "write" call). The final stage of copying the selected files into the new clone is then bound only by how long it takes to copy data from one VM disk image to another. Producing a CS23D clone required approximately an hour.

4.6 Summary

The observations of this section, in particular the 95% and 94% reductions in size for the TNBA and CS23D cases (respectively), and the verification that our slimmed software still functioned correctly, confirms that we met the goals of *minimalism* and *completeness* we set out to achieve in Section 3.1. By applying Lilliputia to two different software packages with distinct construction methodologies, we showed that the VM disk image reduction strategy in Lilliputia can be successful in a broad range of applications.

Chapter 5

Concluding Remarks

Moving software into virtual machines (VMs) can improve their usefulness, by making them portable, self-contained, and easier to back-up and maintain. Reducing their size as much as possible makes moving and deploying them faster and easier.

We showed in detail our implementation of *Lilliputia*, which is an automated system for determining what resources are critical in a VM (by monitoring file access with StatFS), and cloning the important files into a new VM. While monitoring file access with a solution like StatFS is obvious, we show that the implementation details of a system are non-trivial, (e.g., read vs write access, and creation of empty directories). We also showed how monitoring in a network filesystem context is complicated, and solutions to the problems of completeness and minimalism in a network context.

Empirical evaluation of Lilliputia shows its ability to reduce VM image size significantly. Reductions in size of 95% (from 591 to 27 megabytes) and 94% (from 32 GB to 2GB) in the Trellis Network-Attached-Storage (NAS) Bridge Appliance and Chemical Shift to 3D Structure case studies (respectively) means they are easier and faster to move and deploy. This achieves our goal of *minimalism*. Verification of the slimmed VM's correctness confirms that we achieved our goal of *completeness*.

Bibliography

[1] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc., http://aws.amazon.com/ec2/#pricing, 2008.

[2] K. Anwar, M. Amir, A. Saeed, and M. Imran. The linux router. *Linux Journal*, 2002(100):4, 2002.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, University of California at Berkeley, 2009.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, New York, NY, USA, 2003.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] M. Borychowski. Moosefs network file system. `http://www.moosefs.org/`.

[7] S. Burr. How to break out of a chroot() jail. `http://www.bpfh.net/simes/computing/chroot-break.html`.

[8] Canonical, Ltd. Ubuntu server edition jeos. `http://www.ubuntu.com/products/whatisubuntu/serveredition/jeos`.

[9] B. Cantrill. Dtrace on linux. `http://dtrace.org/blogs/bmc/2008/06/30/dtrace-on-linux/`.

[10] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 2. USENIX Association, 2004.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[12] M. Closson, P. Lu, C. Macdonell, and P. Nalos. The trellis nas bridge appliance, November 2009. `http://www.cs.ualberta.ca/~paullu/TrellisNBA/`.

[13] O. Cochard-Labbé, V. Theile, and M. Z. D. Aoyama. Freenas: The freenas server, 2010. `http://freenas.org/`.

[14] W. Cohen. Tuning programs with OProfile. *Wide Open Magazine*, 1:53–62, 2004.

[15] R. Correia. Zfs on fuse/linux. `http://zfs-on-fuse.blogspot.com/`.

[16] D. Geer. The os faces a brave new world. *Computer*, 42(10):15–17, 2009.

[17] Gluster Inc. Glusterfs user guide. `http://www.gluster.com/community/documentation/index.php/GlusterFS_User_Guide`.

[18] S. Graham, P. Kessler, and M. Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):126, 1982.

[19] I. Habib. Virtualization with kvm. *Linux J.*, 2008(166):8, 2008.

[20] J. Howard et al. An overview of the andrew file system. In *Proceedings of the USENIX Winter Technical Conference*, pages 23–26. Citeseer, 1988.

[21] C. Kampmeier. Experimental OpenSolaris Server JeOS. `http://wikis.sun.com/display/Appliance/Experimental+OpenSolaris+Server+JeOS`.

[22] S. Kemp. Using the debian alternatives system. `http://www.debian-administration.org/articles/91`.

[23] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[24] J. Kneschke. Lighttpd. `http://www.lighttpd.net/`.

[25] G. Kroah-Hartman. udev – a userspace implementation of devfs. In *Proceedings of the 2002 Ottawa Linux Symposium (OLS 2002)*, Ottawa, Canada, June 2002.

[26] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 264–275, New York, NY, USA, 1997. ACM.

[27] A. C. Macdonell. Trigger scripts for extensible file systems. Master's thesis, University of Alberta, 2002.

[28] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[29] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[30] P. Oehlert. Violating assumptions with fuzzing. *IEEE security & privacy*, 3(2):58–62, 2005.

[31] Oracle Corporation. Openjdk. `http://openjdk.java.net`.

[32] P. Padala. Playing with ptrace, Part I. *Linux Journal*, 2002(103):5, 2002.

[33] P. Padala. Playing with ptrace, Part II. *Linux J*, 104:4, 2002.

[34] D. Pavlinusic. Fuse - write filesystems in perl using fuse, November 2009. `http://search.cpan.org/~dpavlin/Fuse/Fuse.pm`.

[35] R. Podgorny. UnionFs-Fuse, November 2009. `http://podgorny.cz/moin/UnionFsFuse`.

[36] Qumranet. KVM: Kernel-based Virtualization Driver (White Paper). `http://www.qumranet.com/wp/kvm_wp.pdf`, 2006.

[37] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[38] E. Santos-Neto. bloggerfs. `http://code.google.com/p/bloggerfs/`.

[39] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.

[40] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, A. Surie, D. R. O'Hallaron, A. Wolbach, J. Harkes, A. Perrig, D. J. Farber, M. A. Kozuch, C. J. Helfrich, P. Nath, and H. A. Lagar-Cavilla. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, 11(2):16–25, 2007.

[41] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: a highly available file system for a distributed workstationenvironment. *IEEE Transactions on computers*, 39(4):447–459, 1990.

[42] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[43] E. Shade. Operating systems on a stick. *Journal of Computing Sciences in Colleges*, 24(5):151–158, 2009.

[44] M. Szeredi. Fuse: Filesystem in userspace, November 2009. `http://fuse.sf.net`.

[45] Tuxera. Ntfs-3g. `http://www.tuxera.com/community/ntfs-3g-download/`.

[46] E. Unal. Virtual application appliances on clusters. Master's thesis, University of Alberta, 2010.

[47] E. Unal, P. Lu, and C. Macdonell. Virtual application appliances in practice: Basic mechanisms and overheads. In *Proc. 12th IEEE International Conference on High Performance Computing and Communications (HPCC 2010)*, Melboune, Austrailia, September 2010.

[48] I. VMware. Vmware announces ultimate virtual appliance challenge winners. `http://www.vmware.com/company/news/releases/uvac_winners.html`.

[49] VMware, Inc. VMware vCenter Converter. `http://www.vmware.com/products/converter/`.

[50] VMware, Inc. VMware Virtual Appliance Marketplace. http://www.vmware.com/appliances/.

[51] VMware, Inc. Virtualization Overview. 2006.

[52] N. Wells. Busybox: A swiss army knife for linux. *Linux Journal*, page 10, 2000.

[53] D. S. Wishart, D. Arndt, M. Berjanskii, P. Tang, J. Zhou, and G. Lin. Cs23d: a web server for rapid protein structure generation using nmr chemical shifts and sequence data. *Nucleic Acids Research*, 36:W496-W502:1–7, May 2008.

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