A Method of Merging VMware Disk Images through File System Unification

by

Sarah X. Cheng

S.B., Computer Science and Engineering, Massachusetts Institute of Technology (2004)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2011

Author .....
Department of Electrical Engineering and Computer Science
October 5, 2010

Certified by .....
Stephen A. Ward
Professor
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Master of Engineering Thesis Committee

A Method of Merging VMware Disk Images through File System Unification

by

Sarah X. Cheng

Submitted to the Department of Electrical Engineering and Computer Science
on October 5, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a method of merging the contents of two VMware disk images by merging the file systems therein. Thus, two initially disparate file systems are joined to appear and behave as a single file system. The problem of file system namespace unification is not a new one, with predecessors dating as far back as 1988 to present-day descendants such as UnionFS and union mounts. All deal with the same major issues – merging directory contents of source branches and handling any naming conflicts (namespace de-duplication), and allowing top-level edits of file system unions in presence of read-only source branches (copy-on-write).

The previous solutions deal with exclusively with file systems themselves, and most perform the bulk of the unification logic at runtime. This project is unique in that both the sources and union are disk images that can be directly run as virtual machines. This lets us exploit various features of the VMware disk image format, eventually prompting us to move the unification logic to an entirely offline process. This decision, however, carry a variety of unique implications and side effects, which we shall also discuss in the paper.

Thesis Supervisor: Stephen A. Ward
Title: Professor

Acknowledgments

First, I would like to acknowledge Professor Steve Ward for his support and encouragement, never giving up on me when things were grim. Second, big thanks to Justin Mazzola Paluska for always having an answer to my obscure questions, getting back to me with a quick, timely response no matter what ridiculous hour of day. Third, I would like to mention Professor Ron Rivest for his patience and advice as an academic advisor. Lastly, I cannot forget the countless number family and friends, whether for bouncing ideas off of, or giving me that gentle (and sometimes forceful) nudge to keep going, or even providing a couch to crash on. This thesis paper is a testament to all of your unflinching and selfless support.

Contents

List of Figures

Chapter 1

Motivation and Requirements

1.1 Introduction and Motivation

Consider the IT team of an enterprise, tasked with providing personal workspaces and computing resources to all employees of the company. Traditionally, the team maintains disk images of a standard workstation deployment, which includes a base operating system installation and a few basic applications, such as Microsoft Office. When a new computer arrives, the IT team must manually apply the image onto the new machine. If an employee needed specific applications, such as an architect requesting a CAD application, he would likely need to seek out the IT team, who manages the licenses for all company software. An IT representative would then need to install the application manually on the employee's personal workstation, applying company-specific settings where necessary.

Much of this work is repeated for every enterprise workstation. Is such repetitive manual work necessary? Imagine if the installation and configuration of each application is only done once, and the image of its full configuration is stored centrally in a software "bank." Users can then hand-pick their own pre-configured software suite in a personal workspace, to which their subsequent changes are written.

Under the hood, this would require the ability to merge multiple disk images. The source images are largely identical to each other except for the places where program installations differ, so the merge tool must be able to remove and resolve duplicate

file names while combining any unique directory contents. In addition, the source images must be read-only to normal users so that they cannot arbitrarily edit any images in the software bank. On the other hand, the union image must be read-write to be useful at all.

In this paper, we explore methods of merging read-only source disk images into read-write union images.

1.2 Requirements

Given the vision detailed above, we now enumerate some of the requirements that the final product must meet to realize our goals.

1.2.1 Merging is Done on Disk Image Level via VMware `.vmdk` Images

Foremost, the systems must be merged on a disk image level. More specifically, we loosen our restrictions to require that the sources be normal disk image files, but the unioned result may be anything that looks and behaves as a disk image file to all possible users of the image. From this, we should be able to mount or run the union image directly.

We chose VMware's proprietary `.vmdk` for a couple of reasons. First, VMware is by far the most widely adopted machine virtualization solution, especially among corporate markets. Second, it has built-in snapshotting capabilities via series of delta links (also known as redo logs) upon a base image. This snapshotting ability also conveniently implements our copy-on-write requirement described in the next section, though the delta entries only track block-level changes. Lastly, VMware provides a high-level documentation of their `.vmdk` image format and an API for basic disk maintenance operations, making this project feasible without much reverse-engineering.

1.2.2 Source Images are Read-Only; Union Image is Read-Write

From the description of the inspiration above, we see that the "master" copies must remain read-only, as they must remain reusable after multiple deployments, and that the "personal" copies must support read-write in order for them to be useful. In terms of merging images, this means that the source images must remain untouched while allowing read-write behavior for the resulting union image.

This implies that every union image must maintain its own read-write storage, containing a user's personal changes to the file system. The easiest solution is to make copies of both images, dumping entire directory trees and file contents, and merging directory contents where the directory structures overlap. However, this is a tremendous waste of storage, as a large part of a computer's storage may undergo very few changes throughout its lifetime. This, compounded with the fact that many users will be making copies of the same base images, results in huge amounts of redundant storage in the end.

The other solution is to have the read-write space store the user's personal *changes* to the file system only. Under this design, when the user attempts to access a file, the user's read-write storage is consulted first, in case the user has already made changes to the file; failing that, he must have not altered the file, and thus we can fall back to the base images to look for this file. This type of behavior is called *copy-on-write*, and allows us to store only relevant pieces of data without incurring too much needless redundancy or extra storage.

1.2.3 Support for Merging Multiple Source Images

A useful user environment could hardly be built from two base images. Instead, users should be able to select arbitrary combinations of base images to suit their own needs. This essentially means that our utility must be able to merge any number of images.

It's important to point out here that although our final solution demonstrates a merge of only two base images at a time, the same architecture can be extended to

handle an arbitrary number of source images within a single pass. Our solution is merely a proof of concept with many rough edges, and not intended to be a proper production-level utility.

1.2.4 Low Storage Overhead

This was covered previously, but the extra storage overhead for merging the two disks should be as small as possible. Making copies of entire source disks results in too much needless redundancy and is generally a complete waste of space. Copy-on-write is a more acceptable alternative, though its storage efficiency ultimately depends on the particular implementation itself. In any case, our solution lets VMware handle the copy-on-write for us, so the amount of storage overhead is not under our control.

1.2.5 Performance

Lastly, performance considerations should not be overlooked. Again, since our utility is a proof of concept, we cannot expect the final product to be completely streamlined. That said, online performance should be at the very least acceptable, if not perfect. Offline preprocessing, while much more forgiving performance-wise, still should not take prohibitively or unrealistically long. As it turned out, our solution relies *entirely* on offline preprocessing, so our merged file system is able to run at normal, pre-merge speeds, without incurring any performance penalties.

Chapter 2

Previous Work

There has been much previous work in the area of combining mounted *file systems.* This entire family of unioning file systems performs the merge on a live system, performing lookups and other file system operations on-the-fly. The earliest ancestor of modern unioning file systems, the Translucent File Service, dates back to 1988, while work continues today in the form of UnionFS and union mounts[6]. However, this is not an easy problem to solve, and to this day, every attempt at implementation has been riddled with issues in correctness or complexity. Although unioning file systems are indisputably a much-needed addition to Linux, no single implementation has yet been deemed fit for inclusion in the mainline Linux kernel.

There is surprisingly little work done on merging entire disks or disk images. Our implemention is unique in that it approaches the file system unification problem from a disk-image angle. The ultimate vision is that the final union is not only mountable as a part of a directory tree, but also bootable as an entire operating system. This perspective also frees us from the need to adhere to file system operations and conventions, and lets us attack the problem from a lower level.

In this chapter, we look at the history of unioning file systems, and how some of the concepts and requirements have evolved over time.

2.1 Early Implementations: TFS and IFS

Perhaps the earliest ancestor of unioning file systems is the Translucent File Service, developed by David Hendricks for SunOS in 1988. TFS merges a read-write front layer, to which users' personal changes are written, and a read-only back layer. TFS is so named because the front layer is mounted as the main source file system, while allowing files unique to the back layer to "show through." This also implies that, in the case where a single file name exists in both file systems, the version in the front layer is prioritized. In other words, files in the front layer "mask out" those in the back. This type of approach to unioning file systems, where front layers take precedence over the layers behind them, is known as *stacking.*

TFS creates a type of file called *backfiles* in the front layer. This is basically a lookup cache of files unique to the back layer. To handle lookups, TFS simply looks through current directory entries, then scans the backfiles. The backfiles are guaranteed to be consistent, as the back layer is read-only by definition.

Again, changes to either file system are written only to the front layer. Changes to the front layer behave as a normal file system would handle them. When a user makes a change to a file showing through from the back layer, this file is first *copied* to the front layer, then modified and saved in place. In this way, TFS implements copy-on-write behavior. Deletions from the back layer are handled using *white-outs* written to the front layer, which are essentially entries to "mark" files as deleted. During file lookups, these white-out entries in the front layer simply "mask out" the directory entries in the back layer. Thus, deletions can be allowed in the unioned file system while still maintaining the read-only constraint in the back layer.

In 1993, Werner Almesberger created the successor to TFS, the Inheriting File System[3]. The author claims that it is very similar to TFS, with the main difference being its support for an arbitrary number source file systems, whereas TFS only allowed two. However, he made one key observation that set the precedent for years to come. Both IFS and TFS, having been implemented entirely in-kernel, were becoming much too complex. He suggested that future attempts should take a hybrid approach

that contains both the kernel- and userspace components.

2.2 Union Mounts

The Plan 9 Operating System has a primitive implementation of union mounts via union directories[4]. As a non-UNIX operating system, it does not need to adhere to UNIX standards. As a result, its implementation is comparatively trivial at the expense of some interesting behavior. For example, it does not use white-outs, and does not attempt to merge the namespaces of source directories – when two directory trees contain files with identical path names, the names simply appear twice in directory listings.

BSD union mounts are more advanced than the Plan 9 version and are relatively in line with traditional UNIX semantics. A new directory tree can be mounted above or below an existing union mount as long as the topmost tree is writable. As with TFS and IFS, the topmost tree is given highest priority in namespace resolution and de-duplication. When a directory is first accessed, it is always copied to the top writable layer, a process commonly called *copy-up*. The same copy-up procedure is called when a file is opened in *write* mode only. White-out entries have their own directory entry type, `DH_WHT`, and are written to the top layer either whenever a directory is removed, or only when there is another directory entry with the same name in a lower branch, depending on user configuration. New directories that replace whited-out directories are created by marking them as *opaque* so that lookups will not attempt to skip them.

Linux union mounts are similar in many ways to BSD union mounts. They handle white-outs and opaque entries in much the same way, and file and directory copy-ups behave similarly. File systems are mounted as union mounts by setting the `MS_UNION` flag, and must be mounted either as read-only, or writable with white-out support. Each Linux union mount creates its own `union_mount` structure, which contains pointers to the next layer below (for normal lookups), and to the layer above (for reverse lookups). The differences are in the details of implementation, but architecturally, Linux union mounts borrow much from BSD union mounts.

2.3 UnionFS and AUFS

UnionFS[11] is probably the most well-known of the unioning file systems. Development began in 2003 at SUNY Stony Brook, and continues to this day. A new version, version 2.x, has been proposed, and is the fork under current development.

One improvement that UnionFS offers over previous iterations is its ability to support arbitrary combinations of read-only and read-write mounts. Rather than using a stacking approach as in previous implementations, UnionFS accesses each source file system directly in a "fan-out" manner. Therefore, each source file system is viewed as a branch, rather than as a layer – however, users must still specify a precedence for each branch to resolve naming conflicts. Given files with identical path names, the one in the highest-priority branch always takes precedence, ignoring those in lower-priority branches. In other words, file lookups start at the highest branches and proceed *downwards*. On the other hand, when a user requests a `write()` to a read-only branch, UnionFS searches *upwards* for a writable branch, creates a new file, and *copies up* file contents and metadata. File deletes (unlinks) are executed over *all* writable branches; if the delete occurs on a read-only branch, a whiteout entry named `.wh.<filename>` is created in the next writable branch above.

UnionFS adheres to VFS specifications (detailed in next chapter), essentially looking like any other file system to the kernel. This also means that it defines its own VFS superblocks, inodes, directory entries, and so forth. Each VFS structure provides pointers to private data for file-system-specific use. For UnionFS, this area contains pointers to the objects that a UnionFS file corresponds to in the lower branches. For example, a UnionFS `dentry` for `/foo/bar/` contains UnionFS-specific `unionfs_dentry_info`, which in turn contains a `lower_paths` array. Each item in this array points to the underlying VFS `/foo/bar/` directory entry in a particular branch, as well as to mount structures of the branch. Thus, a file system call to a UnionFS path first passes through VFS, which delegates the call to UnionFS, which in turn calls VFS again for the appropriate underlying file system type.

AUFS (Another UnionFS) is a complete rewrite of UnionFS by Junjiro Okajima,

starting in 2006[5]. Design-wise, it borrows much from UnionFS. It touts numerous features and improvements over UnionFS, such as writable branch balancing (i.e., `write()` always choosing the branch with the most free space). However, the main problem with AUFS is that it is extremely complex, consisting of almost twice as much code as UnionFS 1.x, hardly any of which is documented. AUFS as a whole has been deemed unmaintainable.

2.4 LatticeFS

Yang Su's Master of Engineering project, LatticeFS[10], re-examined the problem of unifying directory trees on a filesystem level. For reasons explained later in this paper, we eventually decided not to use his work, but to start anew at a lower level. Still, we explain his work to provide some background for our discussion of design decisions later on.

LatticeFS combines multiple source directory trees by adhering to configurable conflict resolution rules. The rules are modular, such that users could substitute their own rule by extending a base Policy class. In this way, LatticeFS differs from previous solutions, which all had different, but hard-coded conflict resolution schemes. Another unique feature to LatticeFS is that it does not modify *any* source file system trees; rather, any changes to the combined file system is written copy-on-write to a user's personal storage area *on top* of the source branches. Like previous variants, file deletions from source trees are handled via white-out files in the writable personal storage area.

A typical LatticeFS deployment requires each source directory tree to be mounted, as well as an extra mount point for the unified file system. LatticeFS is built upon fuse (Filesystem in User SpacE) – described more in detail later – to facilitate gluing all of the components together. The main motivation behind fuse is to let developers implement file system operations via userspace programs, rather than having to deal with writing loadable kernel modules. LatticeFS uses fuse to run on *top* of the source file system layer. A typical file access operation involves checking for white-out and

looking for a private version in the personal layer; failing that, it looks through each of the source file systems in turn, obeying any conflict resolution rules.

Two versions of LatticeFS were written, one to focus on performance, and the other to prioritize simplicity and flexibility. The simpler version – called the "runtime" version – resolves conflicts on-the-fly, whereas the "compile-time" version precomputes all conflict resolutions before the union file system is mounted and deployed. However, despite these performance optimizations, LatticeFS ran significantly slower than its constituent source file systems when mounted on their own. At best, it was twice as slow; more commonly, it was around 20 times slower in the performance-centric "compile-time" version and 60 times slower in the "runtime" version. Possible reasons included the python and fuse overhead, but perhaps more importantly, each file access in LatticeFS actually required multiple file lookups – first in personal copy-on-write and white-out area, and then in each of the source branches. Still, it was a very worthwhile and useful proof-of-concept, and highlighted some of the issues and pitfalls to be aware of in our own work.

In any case, whenever there is any online component to run, performance becomes an issue. Programming becomes trickier, and careful considerations must be made in the interest of efficiency. We emphasize this issue now because later we discuss a different approach, in which the merging takes place completely offline, thereby avoiding all of this runtime overhead.

Chapter 3

The VMware Disk Merge Utility

Before we delve into under-the-hood implementation details, we present a high-level description of how our solution to this problem operates. We will also state some of the requirements and properties here, but leave their justifications for the rest of the paper.

At the most basic level, the disk merge utility works by taking in two source VMware disk image (`.vmdk`) files and outputting another image file that contains the union of the two sources. Unlike many of the previous unioning file system solutions, our entire process takes place offline. The utility starts by concatenating the disks themselves together into a single disk. Then, it delves into each disk, examines the contents, and merges them in place.

In the first stage, the virtual disks themselves are joined together. To begin, the utility creates two sets of snapshots for each source disk. Then, it "stitches" one set of snapshots together by concatenating a few lines in a configuration file. Since the snapshotting process invalidates the original source image files, the other set is reserved for acting as the new interface for the source disks.

Now that the union disk appears as one large disk, we can start working with the contents of the disks themselves. First, the source directory trees are copied to the union, creating all the regular files without writing any data to them. The purpose of this step is to merge the directory contents of any directories that are common to both sources, and remove duplicates when two files have identical names and paths.

Now, since we simply concatenated the virtual disks together, most of the data blocks should still be intact from the original disks. At this point, the union disk and the source disks should be almost identical, except for areas overwritten by directory entries or file system data structures. The final step is to use the source snapshots to tell us how to associate the orphaned data blocks back with files. We will also describe a workaround for the overwritten parts of the disk in Chapter 5.

3.1 Input Requirements

There are a few requirements that all source disks must satisfy. They are a result of the simplifications we have made, and it is certainly very feasible to extend our utility to encompass these cases and eliminate the requirements. The rationale for most of these requirements can be found in Chapter 7, where we explain some of the limitations and shortcomings of our disk merge tool.

Currently, the utility can handle only two source images. They must be supplied in the form of VMware disk image (VMDK) files whose descriptor files are separate from actual disk data. The disk images are allowed to contain snapshots, but they must possess the same number of snapshots across all sources. Each disk must be partitioned using `fdisk` or `cfdisk` default values with an MS-DOS-style partition table (63-sector offset, partitions ending on cylinder boundaries). Each file system that is to be merged must reside in the first primary partition of a disk.

We currently only support the ext2 file system format. The desired union block size must be equal to the block size of every source file system. There should be at most one hard link to any file or directory (not including links from `.` or `..`), else post-merge operation of the file system can fail catastrophically.

3.2 Output Characteristics

The resulting disk image, which constitutes the union of the two source disks, is nothing more than a standard VMDK file. In the interest of storage efficiency, the

utility creates a sparse disk, meaning that it does not allocate the size of the entire virtual disk upon creation, but rather lets the image file grow as the on-disk storage grows. Additionally, due to the way VMware snapshots work, the original source images become inaccessible after the merge. Thus, to preserve access to the source disk data, two new snapshots of the source disks are created. Any future requests for the disks should use these new snapshots instead.

The entire merging process takes place offline. The new disk is usable immediately with no need to run any background daemons or other online components. The advantage to this is performance; as a plain disk image file with no runtime logic, it run as fast as any other VMDK image. The main disadvantage inherent to offline processing is that the union disk can only represent a frozen snapshot of the sources at the time of merge. Any future updates to source disks will not be reflected in the union disk.

The current implementation is strictly a merge of on-disk file systems. The union disk may contain OS configuration that is no longer applicable or correct, so it is highly probable that that the union disk will not boot properly. Another inconvenience is that the merge process can introduce file fragmentation to a small fraction of the files. The most serious weakness, however, is the lack of proper support for hard links. All of these issues can reasonably be resolved in future improvements.

Chapter 4

Design Decisions

The different approaches to the problem essentially boils down to the layer at which the merging and copy-on-write is done. Inspired by LatticeFS, the file-system-level approach looks at and compares files as a whole, combining the images above the directory trees. The other idea involves merging at the disk sector level, examining discrete chunks of data on disk. The file-system-level solution requires an online component, while our sector-level design performs the merging offline, with no online components. In this chapter, we examine various pathways and what they entail. All have their advantages and disadvantages, which we shall discuss, and we will provide reasons why we ultimately chose the latter route.

4.1 Strawman Approach

One naive approach is simply to dump file system contents of all source images into the union image, i.e., make copies of all files and directories in each source image. Although this approach is the simplest and by far easiest to implement, it performs poorly at many of our aforementioned requirements. Therefore, a straight file system dump is nowhere near an acceptable solution for our needs.

4.2 A File System-Level Approach

The file-system-level approach mounts both images, then merges the two directory trees above them. It is very similar to LatticeFS, and in fact, the major benefit to this method is that it derives heavily from previous LatticeFS work. However, in doing so, there are many details that must resolved, not all of which are trivial – or even feasible – due to the proprietary nature of VMware products.

4.2.1 VFS and fuse

UNIX is capable of supporting so many different file system types largely thanks to VFS and fuse. They provide clean interfaces that abstract away much of the lower-level differences and details while providing vast flexibility and extensibility to the levels above. It is no surprise, then, that they are major players in our file-system-level approach to the problem.

The VFS[9] (Virtual File System or Virtual Filesystem Switch) layer sits on top of the individual file system implementations. It provides a uniform interface for file system calls that hides the implementation-level differences between disparate file systems. VFS does this by enforcing a common file model to which all file system implementations must adhere. For example, VFS provides a table of function pointers of generic file system operations (such as `open()`, `readdir()`, `stat()`, etc.) that specific implementations must fill in. In other words, to borrow an analogy from object-oriented programming, VFS essentially defines the "interface" or "abstract base class" that specific file system drivers must "implement." Thus, VFS is a "switch" in the sense that it decides which file system driver to delegate the call to, analogous to polymorphism in the object-oriented world.

As previously mentioned, fuse[1] (Filesystem in User SpacE) allows for developers to implement file system capabilities in userspace programs, instead of having to deal with the stringent requirements of writing loadable kernel modules. This helps confine bugs to userspace programs; bugs in a kernel module can potentially lock up the entire kernel. Fuse essentially trades a minimal performance hit for robustness and isolation,

completely justifiable for non-low-level, non-performance-critical operations.

Fuse works by providing two components, a kernel module that sits below VFS at the same level as other file system drivers, and a library that sits in userspace to communicate with the userspace file system utilities. When a user accesses a file within a fuse mount, the corresponding requests are passed to VFS to delegate to the appropriate file system driver. VFS then invokes the fuse kernel module, which in turn interacts with the userspace file system program via the fuse library. Figure 4-1 demonstrates this call flow graphically.

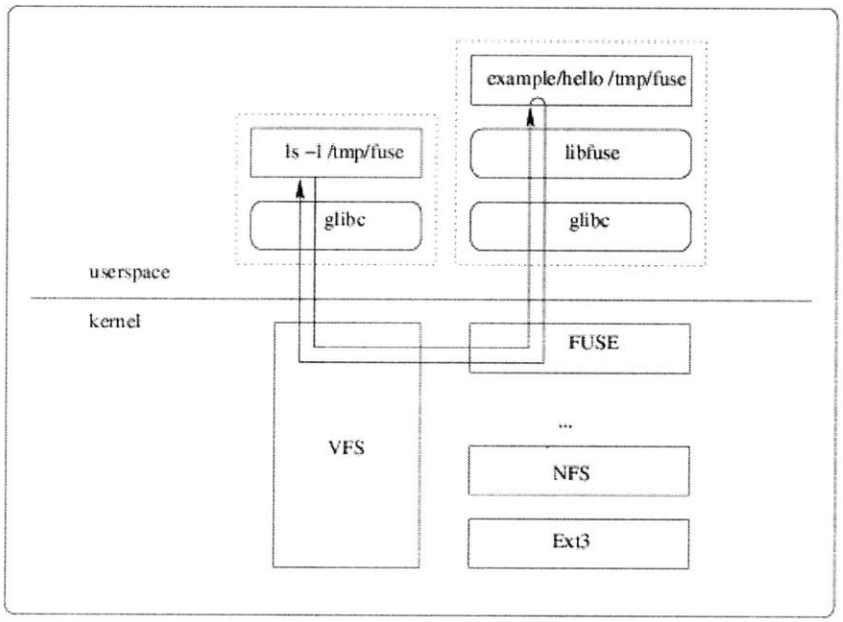

Figure 4-1: The logic flow to a file within a fuse-mounted file system[1].

4.2.2 `vmware-mount`

Many VMware products, including VMware Server and Workstation, include a command-line utility called `vmware-mount`. This utility allows mounting partitions inside VMware images onto the "host" system without having to run any full-scale VM emulation. Essentially, `vmware-mount` provides a translation layer between VMDK and raw disk image formats. Thus, unlike normal operation, under which the image is mounted

and accessed under the guest OS, `vmware-mount` requires that the "host" OS know how to deal with the raw disk image. In other words, given a disk image, the "host" OS must have the appropriate file system drivers, and be able to read and write the file systems contained therein.

As a side note, newer versions of Linux `vmware-mount` also employ fuse to create a loop device that is then mountable as a normal file system. In contrast, older versions used a messy combination of abusing network sockets and loop mounting. This is just one of many examples where fuse helped create a much cleaner, less buggy implementation of the same functionality.

4.2.3 The File-System-Level Design

At the bottommost layer, `vmware-mount` abstracts away details of the proprietary VMDK format and lets us only worry about file system contents, reducing our problem to simply merging source directory trees. At the next layer above, we must present a single file system that reflects the user's changes on top of combined contents of source trees.

To do this, we would use fuse to help present a file-system-like interface to the OS. However, under the hood, our userspace program would redefine file system operations to handle the file system merging, conflict resolution, and personal copy-on-write. These changes would include, for example, simulating `unlink()` operations via adding white-outs, or re-routing `write()` to write changes to a separate personal read-write space rather than overwriting the original file. Most notably, file lookup operations would need to be overhauled to perform a series of lookups in the following order: (1) consult the white-out list, (2) look up the file name in the user's read-write space, (3) fall back to the read-only source trees, in the order specified by the conflict resolution scheme.

The file-system-level merge via fuse is exactly Yang Su's project described earlier, so that part is already done. The next step would be to package the unified file system as a disk image and convert it back to a VMDK format – or at least a format that looks like a VMDK file to the VMware application. Since the file system merge also

takes place on-the-fly, the VMware disk image conversion must also be calculated at runtime. One way to accomplish this is via hooking into `read()` and `write()` system calls between VMware and the host OS – meaning that we would also need to write a host-OS daemon that communicates with the guest OS.[1]

In addition to this complexity, the actual runtime conversion from a file system tree to a disk image might simply be infeasible. During a file access, LatticeFS only looks at very specific slivers in the directory trees corresponding to the file's path. It is extremely difficult to estimate the position of such a sliver with respect to an entire disk without a real-time picture of the entire file system's disk usage. This is a key problem to overcome, as VMware expects a sequential model of a disk, and needs to query specific offsets into the disk. Furthermore, even had we solved the online file-system-to-image conversion problem, the VMDK format is still proprietary and documented only at a very high level. It would require much reverse-engineering to convert a raw disk image format back to VMDK on-the-fly alone.

Figure 4-2 summarizes the steps described above for the file-system-level design. The bottleneck is near the bottom of the chain, which unfortunately appears to be prohibitively infeasible.

4.3 A Disk-Sector-Level Approach

The sector-level approach handles the copy-on-write in the raw data sectors rather than as a layer on top of the file systems. This approach requires dissecting source file systems down to their raw data blocks, then reshuffling these blocks to reconstruct and merge the final file system.

As this level deals with fine-grain disk sectors, we will start by introducing the VMware's VMDK (Virtual Machine DisK) format and introduce a few VMDK-related concepts.

For clarity, as these terms are often interchangeable in practice, from here on *block*

[1]Or one that communicates with LatticeFS running on top of `vmware-mount`, if the disks are accessed using `vmware-mount` rather than through a running VM.

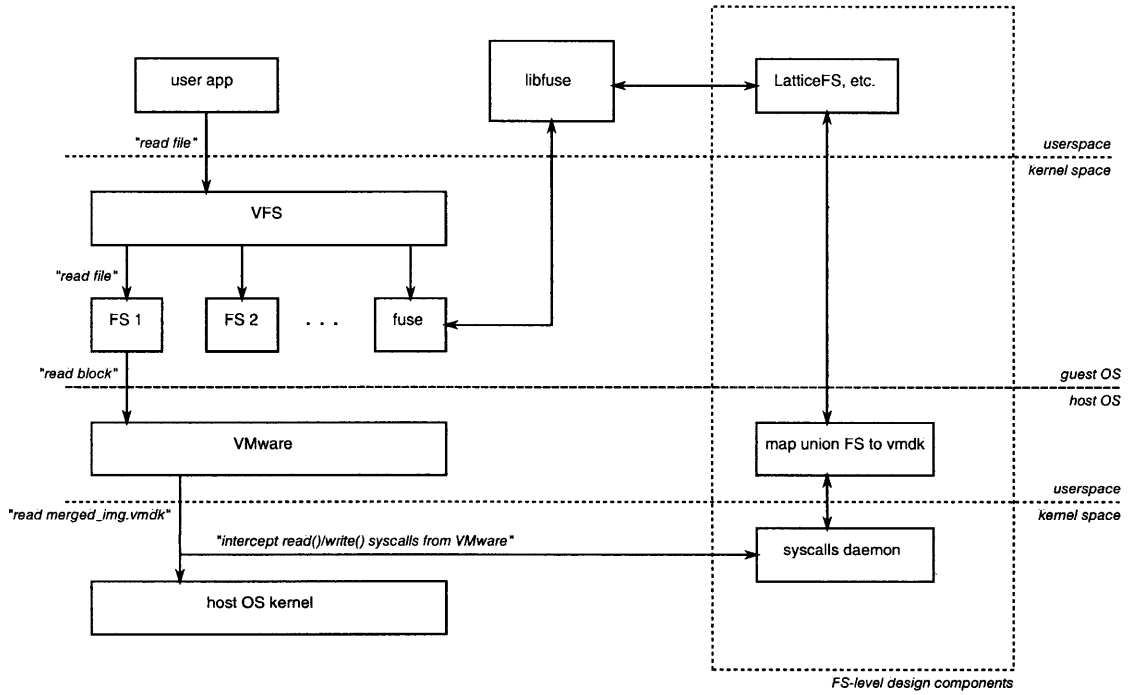

Figure 4-2: The file-system-level design, as compared to normal call flow. The flow on the left describes the path for a standard VMware file access call. The path down the right (through fuse) describes the file-system-level design.

will refer to the basic unit of storage in a file system, whose size depends on format-time configuration settings. The physical hard drive units will be called *sector*s, which are almost always 512 bytes in size.[2] In most sane file systems, file system *block* sizes are multiples of hard drive *sector* sizes.

4.3.1 VMware Image (VMDK) File Format

A normal image file simply contain the raw data within a disk. VMware provides enhancements of its own, and as a result developed their own disk image format: the VMDK (Virtual Machine DisK) format[2]. These enhancements include support for snapshotting, growing the image file size as the virtual disk storage grows (rather than allocating the entire disk at once), and handling arbitrarily large virtual disks on host systems that limits file sizes to 2 GB. Consequently, as more features were

[2]Although newer disks may use larger sector sizes, almost all provide a 512-byte-sector emulation mode for compatibility.

added, the image format became increasingly complex. We will first introduce some basic VMware concepts and terminology, then move on to a basic description of the file format itself.

VMware Image Concepts

VMware supports incremental snapshots through what are essentially redo logs, in which each log entry is called a *link* (or *delta link*). All disk changes are written to the topmost (latest) link. Once a child link is created, all earlier links become unwritable and must remain completely intact. Initially, disks start off with a base image, or base link. When a snapshot is taken, the topmost link is frozen in place, then a new link is created above it to receive all subsequent disk writes. The frozen link represents the state of the disk at the time of the snapshot. A VMDK image is essentially a series of delta links atop a base image, all of which are read-only except for the topmost read-write link.

Each link consists of one or more *extents*. A VMware extent represents a region of storage on virtual disk, which usually correlates to a single `.vmdk` file on the host. Disk data that is stored in exactly one file is described as *monolithic*. Each link contains delta extents that correspond to extents in the parent link. Figure 4-3 demonstrates the horizontal and vertical organization of the VMDK format. In this way, extents operate completely independently of one another; changes in one extent have no effect on contents of another extent.

Extents can also be *flat* or *sparse*, among other types. Flat extents are simplest, and are essentially flat arrays of disk sectors. The downside is that the entire storage space for the disk image must be pre-allocated upon creation. Therefore, creating a 20 GB virtual disk result in a $\tilde{2}0$ GB image file on the host system, even if most of the virtual disk is empty. For this reason, we forgo flat extents in favor of sparse extents, which address this very issue – image sizes start small, but grow as more content is written to the virtual disk. This feature, however, comes at the price of complexity, as directories and tables must be maintained to keep track of newly allocated disk sectors. This is one of the reasons converting raw to VMDK images is non-trivial,

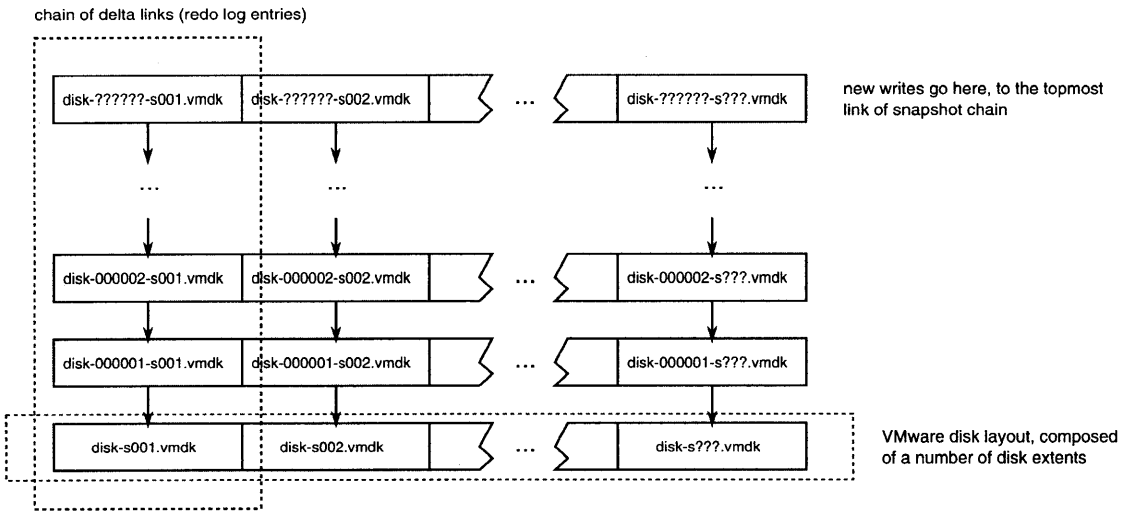

Figure 4-3: The VMware disk image architecture. A horizontal row represents a single snapshot of an entire disk, while a vertical column represents the snapshot (delta) chain of a single extent within a disk.

especially as it must be done on-the-fly.

Descriptions of snapshot delta links and extents can be found in a text-based configuration section known as the *descriptor file*. Essentially, the descriptor file specifies disk settings (such as disk type and disk geometry) and acts as the glue to hold delta links and extents together. The descriptor file can be a text file separate from the disk image data or embedded within an image file; both types of files carry a `.vmdk` extension. For sake of simplicity, our implementation assumes that the descriptor file is separate, although it is certainly not difficult to add support for embedded descriptor files. The next section describes the contents of descriptor files in more detail.

VMDK Descriptor File

A VMDK descriptor file corresponds to a single link along a VMDK image chain. Paths to the link's constituent extents, as well as the location of its parent's descriptor file, are listed here. It is essentially a configuration file for a single VMware snapshot.

The descriptor file is divided into three sections. The first section contains disk image settings, such as the disk type (flat vs. sparse, monolithic vs. split, etc.) or the

parent link file name. The second section is simply a list of extent file names, listed in the order as they appear on disk. The last section provides information about the virtual disk's geometry.

Of particular note, in addition to disk type and parent file name, the first section also contains two mandatory fields, `CID` and `parentCID`. This `CID` value is used by subsequent child links to fill in their `parentCID` fields. Recall that any changes to a disk after taking a new snapshot is written to the newest link only, and any writes to older links invalidates the entire disk. A new `CID` value is assigned to the link whenever the link is opened and modified for the first time. Thus, the `CID` and `parentCID` fields ensure that the parent link has not been tampered with after a child is attached. The `parentCID`s of base links simply use the value `CID_NOPARENT` – `0x0` by default.

The second section is simply a list filenames of the link's extent data. They are listed in order as they appear on disk in the following format:

```
RW 4192256 SPARSE "test-s001.vmdk"
```

The first column specifies access parameters – `RW`, `RDONLY`, or `NOACCESS`. The second column denotes the size of the disk, in physical (512-byte) *sectors*. The third designates the extent type – `SPARSE`, `FLAT`, `ZERO`, and so forth. The fourth column contains the path to the extent data file, relative to the descriptor file. There may be a fifth column for certain extent or disk types, such as flat extents or virtual disks hosted on devices. As we are only dealing with sparse extents, this last column does not apply.

The final section mostly deals with disk geometry, such as CHS (cylinder-head-sector) values that the virtual disk emulates. Assuming the numbers of heads and sectors per cylinder are identical across all source disks (which is likely the case, as users generally use the VMware default values for these), the only pertinent field for us is the number of cylinders. Since the goal is to "concatenate" all the disks together, the number of cylinders in the union disk is roughly equal to the sum of cylinders across all source disks.

4.3.2 The Disk-Sector-Level Design

From the above description of VMDK formats and features, there are two elements of note that are immensely convenient for our needs. First, note that VMware's snapshotting mechanism requires that older links be kept untouched and intact, and instead directs all changes to the topmost, newest link. This exactly describes copy-on-write behavior. Second, the main goal of the project is to combine several disk images. With the descriptor file format outlined above, "concatenating" two disks together is as simple as concatenating a few lines of text. Of course, the union disk would need to be repartitioned for the new size and subsequently reformatted, but a large number of file data sectors on disk should still remain intact.

The problem with both of these "convenient" features, however, is that both the disk concatenation and the copy-on-write are done at a disk sector level – VMware deals with raw disks, after all. Our goal is to merge entire *files* and *file paths*, not *sectors*. Consequently, with copy-on-write and rough merge capabilities already built-in, the sector-level approach shifts the complexity to reconstructing sectors back into sensible files and directory structures.

To do this, file-to-block-numbers mappings for source disks must be interpreted, then translated into mappings for the union disk. Fortunately, since the union disk was formed by concatenating source disks back-to-back, the translations simply involve adding offsets to source block numbers. The offset is also easy enough to calculate – they are nothing more than the sum of the number of blocks for all source disks preceeding the disk in question. Then, the corresponding file on the union disk can re-associate the translated block number back into its block list.

There is one other detail that has been glossed over so far. As stated before, VMware does not allow access to a previous delta link in the chain. This means that if we create child links of source images and naively concatenate them,[3] the source snapshots still become inaccessible and useless. However, with the aid of the VixDisk library, we can create two side-by-side sibling snapshots for the same parent image.

[3]Or worse, if we directly concatenate the source images – which means any user changes would directly modify the sources!

One of the children can then be used to preserve access to the source image, while the other is used for the actual disk merge.

From all this, we construct the following series of steps for the sector-level approach:

1. Create two sibling delta links for each source disk -- one to maintain the original images, and one to be used in the union disk.

2. "Concatenate" source disks by concatenating source extent entries in the descriptor file. Recursively do this for every parent descriptor file until the base link is reached.

3. Expand the first disk's partition to occupy the entire combined disk. Format this new, expanded partition.

4. Mount each source directory trees as read-only (via the appropriate child snapshot from the first step above). Mount the expanded partition from the last step as read-write.

5. Copy each source directory tree in turn to the new partition. Do not copy normal (non-directory) file contents, but do copy file names. The new directory tree should look like a union of all source disks, but with zero-length files only.

6. For every file in every source disk, look up its data block numbers on its *source disk*. Translate each block number with respect to the union disk. If this new block number on the *union disk* is free, associate it with the respective file on the *union disk*.

7. If, however, the translated block number is not free, (overwritten by file system metadata, directory structure, allocated by another file, etc.), allocate a new block on the union disk. Copy all the data from the respective block on the *source disk* into the newly allocated block on the *union disk*.

One huge advantage to the sector-level approach is that the entire merging process is done offline. There are no online components, which means that performance

requirements can be lower and much more forgiving, a problem that plagued LatticeFS and thus would continue to beleaguer the file-system-level approach. In the sector-level solution, the finished version is just nothing more than a plain VMDK image, so there should be absolutely no extra overhead for VMware. The one major downside to an offline merge, however, is that post-merge changes to source images will not be reflected in union images without another explicit offline merge.

4.4 Advantages and Disadvantages

Here we review some of the pros and cons mentioned above.

At first glance, the file-system-level approach seems attractive as there is much previous work to build upon. Previous file system merging utilities such as UnionFS or Yang Su's LatticeFS already solve the conflict resolution and copy-on-write problems for us. Additionally, tackling the issue at the file system level simply makes sense, as the problem itself – merging source directory trees – is a file-system-level one. It handles files as-is, without the need to break them apart, manipulate the files at a block level, and reassemble them. It also allows for independence of source file system formats as long as the host can read them, because the approach operates at the layer above file system internals.

The file-system-level approach works mostly online, rather than offline. This design choice has both its advantages and disadvantages. The main disadvantage is performance – care must be taken to keep the lookup translation overhead minimal. Indeed, LatticeFS's performance was many times worse than that of normal ext3. In offline merges, such as with the sector-level approach, none of the translations are carried out in real time, so performance constraints are much more forgiving. The main advantage to file-system-level solutions is that the source images can be modified post-merge, where changes can readily propagate to the union disk. This can be invaluable in practice where, going back to our original example, a system administrator could constantly update the software image library, so that users always had the most up-to-date software. Offline merges cannot offer this feature, as they are

single-shot, frozen captures of the sources. Furthermore, VMware by convention does not allow modifying parent links with attached child links, and will refuse to read such images.

For all its perceived advantages, however, the file-system-level approach is not viable due to a single roadblock. The problem is that the result of the union must look like a VMDK file to VMware, which queries parts of the image file depending on the sectors requested. It is nigh-impossible to convert a location on a union file system – and possibly one that is constantly being changed and updated – to a physical location on the disk image, much less to do so on-the-fly. There is also the additional problem of converting raw images back into VMDK files, whose format is proprietary and, at best, inadequately documented.

Taking these issues into consideration, we chose to pursue the sector-level implementation. After all, an impasse is still an impasse. For the most part, the major requirements – copy on writes and disk merges – were conveniently already built into the VMware design. As a result, the sector-level implementation is still mostly quite simple in design, with some moderate complexity occurring only in the file and directory tree reconstruction stage.

4.5 ext2 File System

Since we have chosen the sector-level approach, the implementation is now fully dependent on underlying the file system type of the source disks. Thus, we were forced to choose a file system format to demonstrate our proof of concept. We wished to choose something simple in implementation, but nonetheless widely used in practice, so the choices were narrowed down to FAT of DOS/Windows and ext2 of Linux.[4] Ultimately, ext2 won, with the advantage that it is open-source, and optionally includes open-source utilities such as the `e2fsprogs` package. Of particular use is the `debugfs`

[4]In recent years, ext3 has surpassed ext2 in popularity (although slowly moving to ext4). Nevertheless, ext3 is fully compatible with ext2; in fact, ext3 is mostly identical to ext2, only with journaling capabilities. (Ext4, however, is completely different and not fully backwards compatible with ext2 and ext3.)

utility, a userspace ext2/ext3-specific debugger that gives users direct access to many under-the-hood file system operations. Thus, `debugfs` provides a good blueprint on how the ext2 library (`libext2fs`) is used to perform common file system operations.

4.5.1 Basic ext2 Concepts

The basic data structure in the ext2 file system [9] is the *inode*, which contains the metadata for a single file. The inode specifies the data blocks in which the file contents reside, and also provides information such as owner and group IDs and creation, access, and modification times. Inodes are referenced by an *inode number*, which acts as an offset into a table of inodes. The inode table will be covered later as we describe the on-disk layout of the file system.

Recall that UNIX treats every object as a file – normal files, directories, devices, and so forth. Each directory is no more than a file that contains mappings between human-readable file names (including subdirectories and special files) and inode numbers. These directory entries are also the *only* mechanism through which inode numbers are linked to paths and file names.

Multiple paths can correspond to the same inode. Each of these mappings constitute a *hard link* between the file name and the inode. The inode contains a field that keeps track of the number of hard links to it, which is incremented every time a new name is mapped to it and decremented every time `unlink()` is called on one of its associated file names. When the link count reaches 0, the inode is marked as unallocated. Hard links should not be confused with *symbolic links* (or *symlinks*), which are files whose contents contain a path. When a symlink is accessed, the OS simply replaces the path with the target path, and restarts path resolution on this new path. Symlinks do *not* refer to actual inodes, and therefore can point to objects that do not yet exist (such as an NFS file).

4.5.2 Inodes and the `i_block` Array

One of the key functions of an inode is to specify where the file's contents can be found. This information is located in the `i_block` array, which contains block numbers of the file's data blocks. The first 12 entries are *direct blocks*, which directly point to the data blocks themselves. The 13th entry points to an *indirect block* – a block that contains an array of *pointers* to actual data blocks. The 14th entry points to a *doubly-indirect block*, which points to a table of *indirect* block numbers. The 15th and last entry points to a *triply-indirect block*, which as its name suggests, contains an array of doubly-indirect pointers. Figure 4-4 illustrates this set of data structures much more clearly.

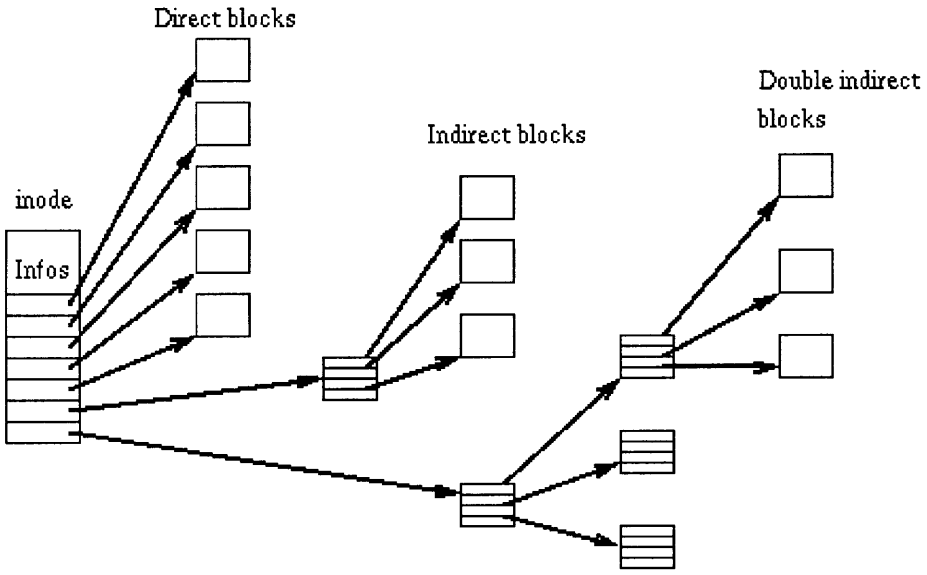

Figure 4-4: The `i_block` array, with indirect block pointers[8].

4.5.3 The ext2 File System Layout

Looking at the partition as a whole, the very first 1024 bytes are reserved for the partition's boot record[7]. The next 1024 bytes are occupied by the *superblock*, which contains basic parameters for the file system, such as block size, blocks and inodes per block group (see the following paragraphs), total number of blocks and inodes,

and so forth. Immediately after the superblock is the *block group descriptor table*, which defines parameters for all block groups, such as locations of the block and inode bitmaps or the location of the inode table. As the superblock and block group descriptor tables are vital to the operation of the file system, both are replicated at the beginning of select block groups throughout the disk. The first block group then begins past this, after the block group descriptor table.

The entire disk is divided into *block groups* of `s_blocks_per_group` blocks each, as defined by the superblock. Therefore, the `s_blocks_per_group` parameter governs the number of block groups in the partition. Each block group contains its own *block bitmap* and *inode bitmap*. These bitmaps are each a single block in size, and keep track of used blocks and inodes, respectively, for its own group. Past the bitmaps resides the inode table, an array of the actual inodes themselves. This table consists of `s_inodes_per_group` total entries, a parameter also defined in the superblock. Finally, the actual data blocks comprise the remainder of the block group.

Figure 4-5 summarizes the general layout of an ext2 partition and the block groups therein.

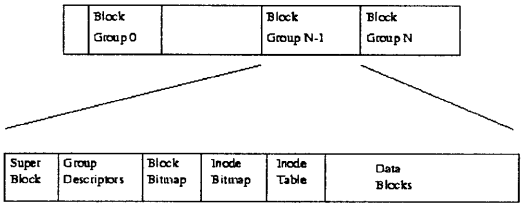

Figure 4-5: The general on-disk layout of an ext2-formatted file system[9].

Chapter 5

Implementation Details

Python was chosen for this project for its ease of development, interactive interpreter (making it quick and easy to experiment with new libraries), and vast array of built-in libraries. As the VMware Virtual Disk library and libext2fs (along with debugfs) are both written for C, we use ctypes to create Python bindings for the libraries. To help programmers work more closely with C libraries, ctypes also provides a framework for specifying function prototypes, declaring structures and unions, and emulating pointers and pointer dereferencing. We use a utility (found in ctypeslib) called codegen to generate these ctypes-styled declarations from the C header and object files.

The VMware Virtual Disk library (VixDiskLib) is provided by VMware to let developers programmatically perform administrative tasks on VMware disk images. The library supports disk operations such as reading and writing, managing snapshots (i.e., creating children), shrinking and expanding, and editing disk image metadata. Of interest to us are `VixDiskLib_CreateChild()` and `VixDiskLib_Attach()`, used to fork our source images into two.

As mentioned before, debugfs offers a blueprint on how libext2fs functions can be used. libext2fs supports almost every basic ext2fs operation at every layer of abstraction, from reading and writing entire files, to manipulating inodes and block bitmaps directly. We will use this library extensively to handle file system reconstruction after the disk concatenation.

Here, we revisit the steps listed in the Disk-Sector-Level Design section in more

detail. For simplicity and as a proof of concept, we only demonstrate merging two disks. This two-disk design can be analogously extended to support multiple disks, with little change in overall design. We also exclusively use sparse extents to conserve storage in our workspace; dealing with other extent types simply requires changing a few flags to VixDiskLib calls, and does not change our implementation very drastically at all.

5.1 Forking Source Images

Before any images can be accessed, `VixDiskLib_Init()` and `VixDiskLib_Connect()` must be called. These are required for VMware's enterprise products, when the VMs reside remotely on large corporate networks, but for our purposes, their arguments should be `NULL` or `0`. Then, each source disk is opened read-only via `VixDiskLib_Open()` with the `VIXDISKLIB_FLAG_OPEN_READ_ONLY` flag, which allows their child links to be created.

Recall that a VMware disk image is composed of a base disk image and a series of delta links corresponding to a timeline of incremental snapshots. Changes to disk are always written to the most recent link. To create a new snapshot (i.e., child link), `VixDiskLib` provides `VixDiskLib_CreateChild()`. In addition, recall that two child links from each source image are required – one to be used for the union disk, and one to maintain that the source images are still accessible. Thus, `VixDiskLib_CreateChild()` is called twice for each source disk, with the `VIXDISKLIB_DISK_SPLIT_SPARSE` flag to create sparse extents. From here on, the term "source" would refer to contents of the second type of link above.

To open the source disks read-only, we call `VixDiskLib_Open()` with the `VIXDISKLIB_FLAG_OPEN_READ_ONLY` flag. However, opening these links locks their parent links as well, which are shared with the links destined for disk union. To work around this, we first open the merge links as standalone links by using the `VIXDISKLIB_FLAG_OPEN_SINGLE_LINK` flag. Then we can manually attach them to the parent using `VixDiskLib_Attach()`.

5.2 Combining Disks via Descriptor Files

The VMware image format is convenient for our purposes in that it allows us to attach disks to one another simply by concatenating lines of a configuration file together. Since descriptor files also reference their parent links, we must generate merged descriptor files for every parent snapshot until the base image, else VMware will refuse to open the disk. This also imposes the restriction that that all source link chains must be equal in depth. This is currently a limitation in our application, but can easily be remedied in future versions by padding child links to be as keep as the deepest chain.

In the first section of the descriptor file, the only pertinent field of note is `parentFileNameHint`, whose value should be the path to its parent's descriptor file. This parent file does not yet exist, and will be generated after the current file is processed. We arbitrarily name parent descriptor files by appending `_p` to its base name. For instance, the parent link and grandparent links of `foo.vmdk` would be named `foo_p.vmdk` and `foo_p_p.vmdk`, respectively. All other fields in this section should just be direct copies of the first source disk's descriptor fields.

The `parentCID` and `CID` fields, although relevant, do not need any special processing, as they are directly copied over from the first source disk's descriptor and its parent's descriptor, respectively. Therefore, the union descriptor's parent CID checks should be consistent if and only if the first disk's checks were consistent to begin with.

The second section is where the meat of the idea resides. This section lists each extent in order that they should appear on the disk. To "attach" the source disks together end-to-end, we simply concatenate the extent entries from each of the sources. Furthermore, we add the sector counts for all extent entries together and keep track of this sum. This value indicates the total size of the union disk, and will be used in the next section.

The third section simply consists of disk geometry settings. Everything in here should be copied as with the first section, except that the number of cylinders (`ddb.geometry.cylinders`) should reflect the size of the new, larger disk. We obtain

the size in sectors of the union disk from the extent listings. From this, the value of `ddb.geometry.cylinders` is calculated to be `total_sectors / (head * sect_per_cyl)`.

Once again, the above must be repeated for all ancestor links of the source images down to the very base images.

5.3 Resizing and Formatting Partitions

At this point, although the two source disks has been joined as one, the partition table, located in the first sector of a disk, still contains the partitioning configuration of the first source disk. Thus, we must resize the partition to make use of the entire virtual disk.

To do this, we draw inspiration from a blog post by user "sudarsan" from VMware Communities.[1] First, we obtain a flat-file (raw) image of the union disk by calling `vmware-mount` with the `-f` flag. Next, we invoke `losetup` to bind a loop device to the raw image file. We then use `fdisk` on this loop device to redefine the partition. To access the new partition, we must first compensate for the 63-sector offset at the beginning of the disk (see next chapter). This can be done by releasing the loop device (`losetup -d`) and rebinding it with a 32256-byte offset into the raw image file. Finally, `mke2fs` is ready to be called on this device. Remember to release the device with `losetup -d` and unmount the flat-file image with `vmware-mount -d`.

5.4 Mounting Source and Union Partitions

In this step, the source and union images are mounted so that the source file systems can be read and replicated onto the union file system. For each mount, a simple `vmware-mount <disk image> <mount point>` would suffice. `vmware-mount` associates a VMware disk image with a block loop device, which is in turn mounted at the specified mount point. To find the loop device backing a certain mount, we

[1] `http://communities.vmware.com/blogs/sudarsan/2008/12/12/short-script-to-create-a-formatted-vmdk`

simply look in `/etc/mtab` for the path to the mount point. In the following steps, we will both be using the loop devices directly and referring to paths within the mounted file systems.

5.4.1 Write Buffering and Remounting

One of the problems that we encountered was block device buffering. Linux generally does not expect multiple writers and readers to the same device, and problems may arise from manipulating a mounted file system (the reader/writer), quickly followed by operating on the device itself. In our case, we experienced this issue between steps 5 and 6 (Sections 5.5 and 5.6), in which the rebuild script could not find many of the files and directories created by the tree copy. There are several ways to force a buffer flush and avoid this altogether, but the easiest is probably to unmount and remount the file system. We can accomplish this by executing `vmware-mount -d` followed by `vmware-mount`.

The issues do not end here, however. Both VMware and `vmware-mount` prevent simultaneous accesses to a disk image by creating lock files upon opening it. However, sometimes `vmware-mount -d` may return before all the locks are fully released, causing the remount to throw an error. For this reason, we always insert an artificial 1-second `time.sleep()` between the unmounts and remounts.

To be on the safe side, we always remount the union file system after every procedure that modifies the disk.

5.5 Copying Directory Trees

The source file systems are now ready to be copied to the union file system. All directories and symbolic links are directly copied, whereas files are given placeholders and left at zero length, with no data blocks allocated to or associated with them.

Python provides an `os.walk()` method in its `os` module that we use to traverse source trees. The path names are split into two substrings, the "source root" (i.e., source mount point), and the "suffix" (rest of the path name). Then, the destination

path name is simply the concatenation of the "destination root" (union image mount point) and the "suffix".

If the file name in question is a directory, we simply call `os.mkdir()` on the destination path. If it is a normal file, we call `open()` on the destination path in read/write mode, then `close()` immediately. In Python, this simply opens then closes the file if it already exists; otherwise, it creates a zero-length file by updating directory entries for the file, allocating an inode, and assigning an inode number. However, this will not allocate any data blocks, as desired. In the case of a symlink, the link destination is read with `os.readlink()` and copied using `os.symlink()`. It is possible to encounter relative symlinks that point above the destination root; however, if this were the case, the symlink would have pointed above the *source* root originally as well, so it was probably a broken link regardless. Devices and other special files in the source trees are ignored and not copied.

To flush any pending write buffers, we remount the entire union file system after each directory tree copy.

5.6 Translating Block Numbers

The idea behind the sector-level merge is to rebuild the file system and directory trees, but to preserve file data blocks. When the union disk was reformatted in step 3 (Section 5.3) above, the data blocks were still mostly there and intact (i.e., intact except for those overwritten by file system or directory metadata), but they were no longer attached to any files. To re-associate files with unattached blocks, we can rely on source file system metadata to provide hints for their data block locations on the union disk.

Because the union disk was created by concatenating source disks back-to-back, the data block numbers do not change with respect to the beginning of each source disk. Furthermore, the block number offset for each source disk with respect to the union disk would simply be the sum of the number of blocks in each previous disk. In other words, the union disk block number b_u, given block number b_n on source disk

n, is given by:

$$b_u = b_n + \sum_{i=1}^{n-1} \text{number of blocks in disk } i = b_n + \sum_{i=1}^{n-1} \frac{\text{size in bytes of disk } i}{\text{blocksize}}$$

From the previous step (Section 5.5), the union file system now contains the full directory tree structures of both source trees. To reattach data blocks to files, we look through each *source* directory tree to tell us which files to work on, then we look up the same name in the *union* tree to find the corresponding inode to write to. We do this by calling `ext2fs_dir_iterate()`, starting from the root inode of the *source* tree. This function takes a directory's inode number and a user-defined callback function as arguments, and executes the callback function on each directory entry therein. Additionally, `ext2fs_dir_iterate()` takes a `void *private` argument that gets passed along to the callback, allowing programmers to supply their own additional arguments. In our case, we set this field to point to the inode number of the corresponding directory in the *union* tree. In this way, the callback function knows about its own place in both the source and union file systems.

For each *source* directory entry, the callback function first looks at the entry's `name` field to determine the file to look up in the *union* file system. Next, to determine its associated inode number, it calls `ext2fs_namei()` on the name with respect to the current working directory in the union (as supplied through the `private` argument above). If the directory entry denotes a subdirectory, it recursively calls `ext2fs_dir_iterate()` to continue crawling through subdirectories within the source tree. Otherwise, if the entry specifies a file, it calls `ext2fs_read_inode()` on the union inode to determine the file's size on the union file system. If the file length is nonzero, then the file has been previously copied, so it is simply skipped. (In other words, in the case of naming conflicts between file systems, earlier disks took precedence.) Otherwise, we follow these steps:

1. For the file in the *union* file system, its inode number should have been previously provided by `ext2fs_namei()`. For the same file in the *source* file system, its inode number can be found in `dirent->inode`. Read the contents of both

inodes into memory with `ext2fs_read_inode()`. Copy the relevant metadata fields – including the `i_block` array – from the source inode to the union inode, and write the union inode back using `ext2fs_write_inode()`.

2. Iterate through the blocks of the file using `ext2fs_block_iterate2()`. This function takes a programmer-defined callback function as an argument, and executes the callback on every block entry associated with the file, including indirect entries. Note that indirect blocks are neither yet active nor processed; the instructions here will also correctly address these issue. The callback function must do the following:

   (a) Translate the block number into the union file system block number using the formula above.

   (b) Call `ext2fs_test_block_bitmap()` to query the union disk's block bitmap and check whether the block number has already been allocated. If so, see next section, and perform the steps therein before continuing. Otherwise, use `ext2fs_mark_block_bitmap()` to mark the union block number as allocated. Make sure to decrement the block group's free blocks count, and to mark the block bitmap and superblock dirty with `ext2fs_mark_bb_dirty()` and `ext2fs_mark_super_dirty()`.

   (c) Update the block number entry by reassigning the new value to `*blocknr`. `blocknr` is the pointer to the block number entry, supplied as an argument to the callback by `ext2fs_block_iterate2()`.

   (d) Return `BLOCK_CHANGED` from the callback to force the new block number to be written to disk.

Be sure to flush the write buffer after each pass by remounting the union file system.

5.7 Handling Overwritten Data Blocks

This step should only be conducted when a translated block is found to be already allocated (see previous step, Section 5.6). To do this, we simply allocate a new block in the union disk and copy the contents from the conflicting block on the source disk to the newly allocated block:

1. Allocate a new block on the union disk with `ext2fs_new_block()`. `ext2fs_new_block()` also takes a `goal` parameter that specifies where to start searching for free blocks, which in our case should the end of the original partition boundary of the first source file system. We can easily determine the location of this boundary by looking at the `s_blocks_count` parameter in its superblock. See the justification below.

2. The location of the corresponding source block is simply the pre-translation block number. Read the contents of this source block with `io_channel_read_blk()` from the *source* file system.

3. Copy the contents to the newly allocated block on the *union* disk with `io_channel_write_blk()`.

Block conflicts can occur if the block has been overwritten with file system metadata (e.g., bitmaps or inode tables), directory data from the rebuilt directory tree (from step 5, Section 5.5), or file data belonging to an earlier overwritten block. The first two cases are unavoidable, but the last case should be prevented as much as possible. Moving a conflicting block to a location that will cause future conflicts simply perpetuates the problem. Therefore, each time we allocate a new block on the union disk and copy the corresponding source content over, we must choose a block number to which future blocks are unlikely to translate.

Most disks today retain the cylinder-head-sector interface for compatibility, even though the values have little correlation to the actual physical geometry of the disk. For this reason, many disks today do not actually end on cylinder boundaries. Since most modern disk partitioning utilities still adhere to the old DOS convention of

aligning partitions to cylinder boundaries, there is often unpartitioned space at the end of a disk. When multiple disk images are concatenated together to form the union image, the partition therein is expanded to include this unused space. Thus, this space is writable in the union disk but had been inaccessible in the source disk, so there is no chance of future conflict. For this reason, we always look there for new blocks to allocate.

Chapter 6

Performance Evaluation

6.1 Benchmark Test Setup

Here, we seek to validate two of our earlier claims. First, we wish to show that, because the merge output is a normal VMware image file, there is negligible difference in file access times between pre- and post-merge images. Second, we would like to ensure that the offline merge process itself does not take prohibitively long to complete.

We created two 6 GB VMware disk images to demonstrate the functionality of our merge utility. Both were initially manually partitioned using `fdisk`, creating a single large partition that takes up the entirety of each disk. No space was allocated for swap partitions, nor any other data partitions. We formatted both partitions as ext2 with the default 4096-byte block size, and installed Linux Mint 9 32-bit onto both using identical settings. To simulate the idea of an application bank, we installed Chromium web browser onto one disk image and VLC media player onto the other. Each setup used about 2.5 GB out of an entire 6 GB disk. All subsequent I/O tests are performed on these images.

We created another set of smaller images for the offline merge tests only. These were 4 GB in total available disk space, only < 100 MB of which were used. They contained no operating system installs, and were instead composed of two simple directory trees of a few test files. These images were made purely as a basis for comparison against the above Linux images.

The offline merge was timed using Python's built-in `cProfile` module, which measures the CPU time spent in each Python function. This gave us a useful breakdown on the individual running times of each step of the process. This test was carried out twice, once on each of the two setups above.

File access times were analyzed using an I/O benchmark script by Antoine Pitrou from the Python 3 SVN repository.[1] Test files were either 20 KB, 400 KB, or 10 MB in size – to represent small, medium, or large files – with two types of files for each size, plaintext and binary. The script looked for these test files at the beginning of each run and generated them if they were not found. Text files were produced from hardcoded strings within the script itself, while binary files were generated from calling `os.urandom()`. Users can provide command-line options to specify which tests to carry out. To reduce the amount of variation due to background disk usage, each file I/O test was looped until at least 1.5 seconds had elapsed.

We ran each set of tests twice on the Linux install image, once using `vmware-mount` on the host system, and once from within the guest OS of a running VMware image. The script tested for a variety of benchmarks, but we will only show a subset of the data. We will look at file reads on entire files at a time, using the aforementioned sizes of 20 KB, 400 KB, and 10 MB of files. We will also compare file overwrites and appends, written in chunks of 1, 20, or 4096 bytes at a time. For file reads, we tested for what we called pre-merge reads, post-merge reads, and post-merge rereads. Pre-merge reads are simply reads straight from a source file system, before any of the merging process takes place. Post-merge reads are reads of the same file right after a disk merge, before any changes have been written to it. (Recall that the script will only generate new test files if they do not already exist.) Post-merge rereads, on the other hand, denote reads that take place after post-merge writes to the test files. For file writes, we tested for overwrites and appends, both pre-merge and post-merge. Appends are simply writes to the file from wherever the file pointer is at the time, while overwrites seek to the beginning of the file before writing. Pre-merge writes

[1]The script was written by Antoine Pitrou for a Python 3 development bug report, found here: `http://bugs.python.org/issue4561`. The script itself was pulled from `http://svn.python.org/projects/sandbox/trunk/iobench/`, revision #85128.

are analogous to pre-merge reads. Post-merge writes take place after all post-merge reads have completed, but before the post-merge rereads.

All of the tests described here were conducted on a netbook with dual Intel Atom N330 processors, 2 GB of RAM, and a 5400 rpm SATA hard drive, running the 64-bit version of Linux Mint 9 Isadora. All virtual machines were launched through VMware Workstation 7.1.2, emulating a single-core 32-bit processor with 1 GB of RAM and a SATA hard drive. The VMs were all running Linux Mint 9 Isadora 32-bit. The hardware specifications are fairly low-end, and we hope to prove that our merge utility can run well even low-powered systems such as netbooks.

6.2 Offline Merge Results

Figure 6-1 shows the performance breakdowns of the the small (< 100 MB) test file system, as generated by `cProfile`. Extraneous lines, such as primitive library calls, have been omitted for legibility. The entire run completed in 64 seconds of CPU time. This number may seem fairly worrisome for such a small file system, until we look at the individual function breakdowns. Of the 64 seconds, 41 seconds were spent on step 3 (Section 5.3), resizing and formatting partitions with `fdisk` and `mke2fs`. The time spent flushing buffer caches and unmounting VMware images (`vmware-mount -d`) cannot be ignored, either – another 12.5 seconds total were spent purely on unmounting in steps 5 through 7 *alone* (Sections 5.3, 5.6, and 5.7), including the 1 second `sleep()`s in between. This comes out to an average of about 3.1 seconds *per* unmount operation! The actual tree copy (step 5) took only 0.17 seconds in comparison, while the file system rebuild (steps 6 and 7) took just 1.9 seconds.

Contrast these to the run using Linux Mint images (about 2.5 GB of data), shown in Figure 6-2. These results are likely more representative of practical use. The merge process required 664 seconds of CPU time to complete, or just over 11 minutes. Of this, 365 seconds were spent on steps 6 and 7, rebuilding the files from blocks. Another 201 seconds were spent on step 5 in recursively copying both directory trees. This comes out to a total of 9.5 minutes spent reconstructing directory tree contents, not

```
        Running Time Breakdown for Merge: 100 MB File System
      23918 function calls (23588 primitive calls) in 64.263 CPU seconds
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001   64.263   64.263 runmerge.py:76(main)
     1    0.001    0.001   41.128   41.128 fixpart.py:7(do_fixpart)
     3    0.000    0.000   11.133    3.711 runmerge.py:71(remount_dest)
     2    0.000    0.000    1.906    0.953 rebuild.py:172(do_rebuild)
     2    0.003    0.002    0.170    0.085 copytree.py:42(do_copytree)
   2/1    0.003    0.002    0.018    0.018 mergevmdk.py:36(do_merge)
     1    0.001    0.001    0.005    0.005 runmerge.py:13(get_devices)
```

Figure 6-1: `cProfile` results of merge operations on the small test file system. The numbers in columns are in seconds of CPU time.

counting the 26 seconds spent simply remounting the union. With these disk images, resizing and formatting partitions (step 3, from Section 5.3) only took 62 seconds in comparison. This is consistent with the test file system run, as the total capacities of these disks were 50% larger, so consequently, `mke2fs` had approximately 50% more block groups and inodes to write.

```
        Running Time Breakdown for Merge: Linux Mint Install
      13720544 function calls (12558870 primitive calls) in 663.768 CPU seconds
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001  663.768  663.768 runmerge.py:76(main)
     2    0.000    0.000  364.623  182.312 rebuild.py:172(do_rebuild)
     2    9.230    4.615  200.819  100.410 copytree.py:42(do_copytree)
     1    0.001    0.001   61.635   61.635 fixpart.py:7(do_fixpart)
     3    0.000    0.000   25.758    8.586 runmerge.py:71(remount_dest)
   2/1    0.003    0.002    0.019    0.019 mergevmdk.py:36(do_merge)
     1    0.001    0.001    0.003    0.003 runmerge.py:13(get_devices)
```

Figure 6-2: `cProfile` results of merge operations on the small test file system. The numbers in columns are in seconds of CPU time.

6.3 File Operations Performance

Figures 6-3 and 6-4 show the results for the `iobench.py` benchmark test. Figure 6-3 shows the measurements from the host OS test, using `vware-mount` to access disk image contents; Figure 6-4 shows results of the test conducted within the guest OS of a running virtual machine. The series in red and orange represent pre-merge values, whereas the series in blue and purple reflect post-merge data. Most of these series overlap or come close, especially in the host OS tests, signifying that there is

no appreciable difference between pre- and post-merge I/O performance.

Since system resources must be diverted to running the VMware application, the data from the guest OS tests might have been impacted by background hardware activity. This is especially true given that the benchmarks were run on a low-powered netbook, and the vast degradation in throughput numbers confirms this. It might also explain the wide variation in the guest OS statistics compared to the relatively consistent host OS test results.

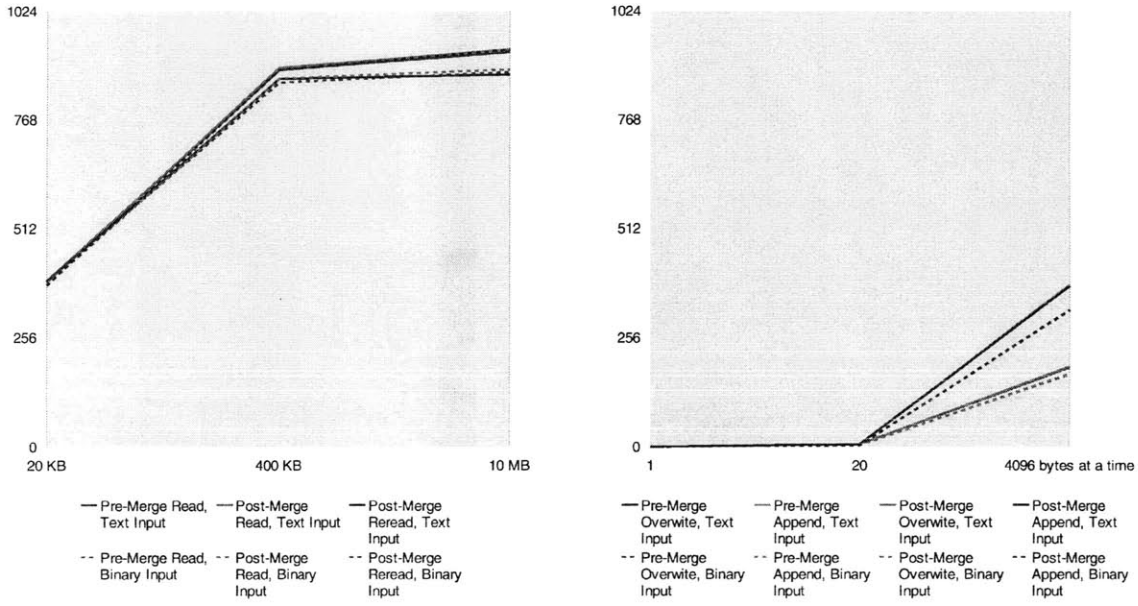

Figure 6-3: Pre- and post-merge performance comparisons (in average MB/s) for read and write operations, as conducted on the host OS. The VMware images are mounted using `vmware-mount`.

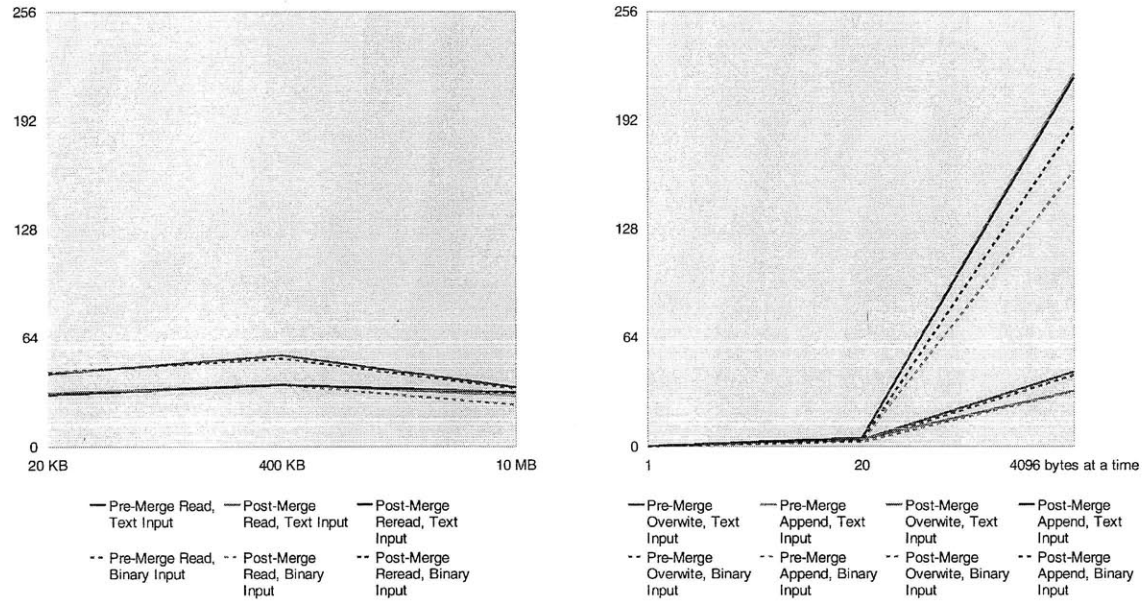

Figure 6-4: Pre- and post-merge performance comparisons (in average MB/s) for read and write operations, as conducted on the guest OS. Due to overhead from running the VMware application itself, these measurements may be less accurate.

Chapter 7

Issues and Limitations

As with any design, there are advantages and there are drawbacks. Some of the limitations here were due to conventions or design decisions in components we were depending on. Others were due to simplifications we had made for this "proof of concept", so that viable but more complex solutions are attainable. Here we analyze some of the issues and possible workarounds, if applicable.

7.1 Heavy Dependence on File System Configuration

First, the merge utility currently has no support for file systems other than ext2. This would otherwise be fine – ext2 was chosen as a proof of concept, after all – had the file system reconstruction not been such a major part of the entire process. In other words, support for other file systems is more than a mere "plug-in"; rather, it becomes a significant part of the implementation. Furthermore, file system implementations can be varied enough that it is difficult to formulate a design that lends itself to a plug-in architecture. Still, this utility demonstrates that merging systems in a VMware disk image level is still possible, albeit messy.

As mentioned before as a disadvantage of the sector-level approach, the current design limits the union disk and all of the source disks to the same file system format.

At best, each combination of mixed formats would require a separate implementation to deal with the differences. This may become a problem in a large network with disparate systems, for example. Additionally, in the case of ext2, the implementation also restricts source and union file systems to very specific block sizes – in our case, requires both source and union file systems to utilize 4096-byte blocks.

7.2 Source Disk Requirements

Some limitations in the design impose requirements that the source disks must satisfy. For example, in the step that merges VMDK descriptor files, the number of heads and sectors per cylinder should be equal across all source virtual disks. It is possible that these parameters could be adjusted without needing to tweak the data portions of the disk images, but in any case, this problem is most likely irrelevant as disks are usually created with VMware's default disk geometry values.

Recall that disks are merged simply by appending raw image data together. Since we used a block size of 4096 bytes (i.e., 8 sectors), the number of sectors for each source partition must be a multiple of 8 – else, data blocks on later source disks cannot be read as block boundaries will be misaligned. However, this is seldom a problem in practice, because the numbers of heads and sectors per cylinders are often multiples of 8 themselves.

More significant limitations arise from an old DOS partition table convention, which imposes an offset of at least 63 sectors in the first primary partition and every logical partition, but does not require the offset for any other primary partitions on the disk. This can be seen in the following example partition table in Figure 7-1 (notice the column labeled "`Start Sector`"):

Many partitioning utilities, such as `fdisk` or `cfdisk`, defaults to the minimum 63 sector offset when creating a new partition. As 63 is not a multiple of 8, care must be taken once again to avoid block alignment issues. Merging partitions will not be an issue *as long as all partitions are multiples of 4096 bytes* ***and*** *whose pairwise*

#	Flags	---Starting---- Head	Sect	Cyl	ID	----Ending----- Head	Sect	Cyl	Start Sector	Number of Sectors
1	0x00	1	1	0	0x83	254	63	11	63	192717
2	0x80	0	1	12	0x07	254	63	4178	192780	66942855
3	0x00	0	1	4179	0x83	254	63	5484	67135635	20980890
4	0x00	0	1	5485	0x05	254	63	19456	88116525	224460180
5	0x00	1	1	5485	0x83	254	63	5877	63	6313482
6	0x00	1	1	5878	0x83	254	63	7836	63	31471272
7	0x00	1	1	7837	0x82	254	63	7968	63	2120517
8	0x00	1	1	7969	0x07	254	63	19456	63	184554657

Figure 7-1: Sample partition table, showing the DOS-style 63-sector offsets.

differences between offsets are multiples 8.[1] However, using the above disk as an example, alignment problems will arise when merging the first primary partition with another primary partition, as the first partition will exhibit a 63-sector offset while the other partition will not.

Unfortunately, not all partitioning utilities default to 63-sector offsets. In our implementation, we assume that source disks are formatted using `fdisk` or `cfdisk`, and each disk contains a single primary partition that spans the entire disk. Future implementations can sidestep this assumption by reading the partitioning table and appropriately adjusting block number offsets before the rebuild stage.

7.3 Incorrect Support for Multiple Hard Links

Since the entire directory tree is blindly copied in step 5 (Section 5.5) without looking at its contents, our utility does not properly handle hard links. Rather, it creates several different inodes that point to the same data blocks, rather than using a single inode and incrementing its link count. Thus, deleting one of the files will also free all of its data blocks, which can be catastrophic as other inodes are still referencing the same blocks!

Support for multiple hard links is also a problem in both BSD and Linux union mounts, so this issue is not a unique one. However, their approaches fail more grace-

[1]Note that both disk and partition sizes tend to be multiples of 4096 bytes, so there usually are extraneous unusable sectors at the beginning and end of both disks and partitions as a result of any awkward offset sizes.

fully than our current implementation[4]. One way to improve hard link handling is to build a reverse mapping of inodes to file names on the source file system prior to the directory tree copy, so that the corresponding inodes on the union file system can be mapped to the same files. This, of course, introduces extra complexity to the program, but fixing this issue should be highest priority in any future work.

7.4 Minor Issues and Future Improvements

When remapping block numbers to a union disk, the method we used to handle overwritten data blocks can lead to file fragmentation. In the current implementation, these blocks are handled by searching for and allocating free blocks starting from the end of the original partition where they are least likely to cause future conflicts, then copying contents of the source data blocks to these locations. However, this also scatters file fragments into non-contiguous parts of the disk. Defragmenting the disk post-merge may help, but moving blocks around likely increases the sizes of delta link files. A better solution may be a "two-pass" approach to the block number translation phase. In the first pass, source block numbers are translated to union block numbers as usual, except overwritten block numbers are only marked for repair. The possibility of future conflicts is no longer an issue, as all non-conflicting blocks have already been accounted for. Then, the second pass handles overwritten blocks much the same way as now, except free blocks are allocated starting from the same block group, rather than from the end of the original partition. By placing fragments of files closer together, we can potentially reduce seek times.

As discussed in the previous chapter, a current limitation is that all source images must contain the same number of snapshots. In other words, the delta link chains across all source disk images must be identical in length. This limitation is a result of the VMDK descriptor file format, which references both the parent descriptor file *and* every delta extent of the current link. One simple method around this limitation is to pad every source chain with repeated snapshots, until all are as long as the longest source chain. With sparse extents, the storage overhead for these extra snapshots is

minimal.

Another problem is that a mere disk merge, as presented, cannot render the union disk immediately usable in the manner envisioned in the introduction. The disk image merge is only a step towards that goal. There are still many other outstanding concerns that must be addressed. For instance, file metadata such as user and group IDs may not be set up correctly, as the relevant entires may be missing from `/etc/passwd` and `/etc/group`. In addition, VMware may regard the merged image as a new hardware device, which can create problems when `/etc/fstab` refers to disks by UUID and cannot find the old hard drive. Moreover, the disk image merge makes no attempt to set up device and other special files correctly. Issues such as these will have to be handled on the guest OS level, and is beyond the scope of this project.

Chapter 8

Conclusion

The idea of a unioning file system is not a new one. Various iterations have existed for over 20 years. However, most solutions have had at least one major weakness or flaw, and not one has yet been deemed sufficient for inclusion in the mainline Linux kernel.

In this project, we attempted to explore a different version of the same problem. We sought to merge discrete sets of files not just as mounted file system, but as entire disk images. We also chose to use a completely offline merge rather than an online solution, which is currently the prevailing trend. This has a number of implications that sets it apart from the existing methods today. The most notable difference, for example, is that our union disk reflects *frozen* snapshots of the sources. In other words, any post-merge changes to a source disk will not be reflected in the new disk. This is sufficient for applications such as static "software banks" (as envisioned in Chapter 1) or LiveCD images. On the other hand, one of the major draws of real-time file system merging is the ability to propagate source file system changes to the union. This is useful when source file systems that are constantly updated, such as database systems or web services. Thus, we envision our approach as serving a different niche from the existing live file unioning implementations; it will be useful to an entirely different domain of applications. As a result, we do not believe that it will compete with UnionFS and union mounts, but rather co-exist alongside them.

Bibliography

[1] Filesystem in userspace. `http://fuse.sourceforge.net/`.

[2] Vmware virtual disks: Virtual disk format 1.1. Technical report, VMware, Inc., 2007.

[3] Werner Almesberger. Re: Union file system. Mailing List Archives, available at `http://marc.info/?l=linux-kernel&m=104931921815733&w=3`, November 1996.

[4] Valerie Aurora. Union file systems: Implementations, part i. *LWN.net*, March 2009.

[5] Valerie Aurora. Unioning file systems: Implementations, part 2. *LWN.net*, April 2009.

[6] Valerie Aurora. A brief history of union mounts. *LWN.net*, July 2010.

[7] Dave Poirier. The second extended file system: Internal layout. `http://www.nongnu.org/ext2-doc/ext2.html`.

[8] Theodore Ts'o Rémy Card and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[9] David A. Rusling. The file system. In *The Linux Kernel*, chapter 9. 1999.

[10] Yang Su. File system unification using latticefs. Master's thesis, Massachusetts Institute of Technology, May 2009.

[11] Charles P. Write and Erez Zadok. Kernel korner - unionfs: Bringing filesystems together. *Linux Journal*, December 2004.