File System Unification Using LatticeFS

by

Yang Su

B.S., Massachusetts Institute of Technology (2008)

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

Master of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author . . . . [illegible] . . . .
Department of Electrical Engineering and Computer Science
May 8, 2009

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

Accepted by . . . . [illegible] . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

File System Unification Using LatticeFS

by

Yang Su

Submitted to the Department of Electrical Engineering and Computer Science
on May 8, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science

Abstract

LatticeFS is a namespace unification system designed to merge multiple source file systems into a single working file system. LatticeFS can be used to merge multiple software package directories, work with multiple file systems as if they are one, and share a single storage medium among multiple machines. On a high level, LatticeFS takes as input an arbitrary number of file system paths, and mounts a new virtual drive that will appear to the user as a union of the input file systems.

Of course, attempting to combine multiple file systems will inevitably be met with conflicts. Situations in which multiple input file systems contain files/directories with the same name will be common in large systems; which file/directory should the user be exposed to in this case? Previous work such as UnionFS solved the problem by giving each input file system a strict priority value, and when a conflict occurred, the file/directory with the highest priority was the one shown to the user. In LatticeFS, we have introduced a plug-in system in which different strategies for resolving conflicts can be easily swapped in and out; additionally, handlers for special file types can also be "plugged" into the system. This paper describes and evaluates all aspects of LatticeFS in detail.

Thesis Supervisor: Stephen A. Ward
Title: Professor

Acknowledgments

First I'd like to thank Professor Steve Ward, who, with his invaluable wisdom and insight, guided the LatticeFS project every step of the way. I'd also like to thank Justin Mazzola Paluska for the incredible amount of help and guidance that he has given me throughout the year, and Hubert Pham for taking the time to proofread/code review my work.

I'd like to thank all of my friends and family for their support over the years, without which MIT would have been insurmountable.

Contents

List of Figures

Chapter 1

Introduction

LatticeFS is a new file system unification tool that seeks to solve the problems that plague other systems of the same type. The motivations behind file system unification are many; for example, if a user wishes to work with different sets of files and directories which are inconveniently located on many different volumes, he or she can simply use our system to create a virtual volume that contains all of the required files and directories conveniently located in one place. LatticeFS also introduces the idea that since some special file types (tar archives, virtual machine images) are hierarchical just like traditional file systems, it should be possible to merge the contents of these files during file system unification. In this paper, we explore the technical details and rationale behind the implementation of LatticeFS, its strengths, weaknesses, performance, and future work.

1.1 What is File System Unification?

A file system unification tool is something that can take as input multiple file systems, and output a new file system that is a unified, or merged, view of the contents of all of the input file systems. One difficulty in file system unification is that the input file systems may be either read-write or read-only. Of course, we wish to give the user the impression that the virtual file system is write capable; thus, most file systems of this type use the idea of copy-on-write. The idea is that whenever the system needs to modify a file within a a read-only directory, it copies the file into a write-capable file system, and modifies it there; as long as the modified copy is of higher priority than the original copy, the user should see no difference. Another difficulty with file system unification is conflict resolution. For example, if an input file system has some file/directory with the same absolute pathname as that of another input file system, then it is unclear as to which one should be exposed to the user. LatticeFS differs from other file system unification tools primarily in this area.

1.2 Other File System Unification Tools

As stated in the previous section, conflict resolution is the main hurdle to clear when building a file system unification tool. UnionFS[1], arguably the most established unification tool, adopts the idea of a priority queue to resolve conflicts. Within UnionFS, each input file system is given a priority value, and, when a conflict occurs, the file/directory from the highest priority input FS is the one shown to the user.

The Union Mount system[3] is another file system unification tool and works as follows: the first mounted file system behaves as normal, and every subsequent file system mounted on top of it simply adds to the original file system its own contents. When a conflict occurs, the newly mounted file system takes precedence. Thus, we can think of it as a priority queue with the highest priority given to the newest mounted file systems.

The problem with UnionFS and Union Mount's approach is that the newly created file system may not work correctly; for example, if some program depended on a set of files that were not exposed to the user (as a result of lower priority), then the program would not behave as expected. To solve this problem within LatticeFS, we have implemented a plug-in architecture that supports easy swapping of conflict resolution techniques from simple priority queues like UnionFS to one in which the user is prompted when a conflict is met.

Another difference between LatticeFS and previous implementations of file system unification is that all input FS are treated as read-only in LatticeFS. Of course, LatticeFS allows user modification, but any changes are kept in a separate storage directory. Additionally, LatticeFS allows for a set of special file handlers that resolve certain file types differently. For example, if we had two "conflicting" archive files, an easy way to resolve this conflict is to create a new archive file with the contents of both conflicting files.

1.3 Thesis Outline

Chapter 2 will be on related work. Chapter 3 summarizes the original design goals set for LatticeFS, and the challenges met along the way. The subsequent two chapters address the final design and implementation of LatticeFS, and Chapter 6 discusses its performance. Finally, Chapter 7 discusses future work and ideas for improvement.

Chapter 2

Related Work

File system unification has existed since at least the Translucent File Service (TFS) [4], written around 1988 for Sun OS. The first similar system for Linux was the Inheriting File System (IFS) [5]. Then there was UnionFS (2003), Union Mounts (2004), and AUFS (2006) [2], as well as FUSE (File system in User SpacE) [6]. In this chapter, we will briefly examine these previous attempts at file system unification, and their impact on LatticeFS.

2.1 TFS

TFS, or Translucent File Service, was developed so that some set of files could be shared among many people. During the time that it was developed, storage space was not nearly as abundant as it is now, so instead of replicating a large set of files that all users within a given network needs on each individual machine, TFS was developed so that these files can be used by everyone on the network.

Within TFS, shared files that everyone has access to would coexist among files private to each user. If the user wishes to modify a shared file, then the file is copied over to the user's private directory, and modified there. This feature is called *copy-on-write*. TFS utilizes two priority levels: a higher priority for the user's private data, and a lower priority for the shared data. Any duplicates within the two sets of data will be resolved by showing the user the copy of the file within the user's private data. Thus, when a file is modified through the use of copy-on-write, the modification is seen by the user. For deletions, TFS utilizes an object called a *white-out*, which is something that designates a file as non-existent. Thus, if a user wishes to delete /a/b/c that exists within the read-only shared data section, TFS simply creates a white-out for /a/b/c that designates it as being deleted. From then on, the user is no longer shown the "deleted" file.

2.2 IFS

IFS, or the Inheriting File System, was inspired by TFS. It was designed such that instead of being able to support two unified file systems like TFS's shared/private data, it could support an arbitrarily large number of file systems. Conflict resolution was also similar: the input file systems were all given a priority level, and the file with the highest priority would be the one exposed to the user. IFS also utilized white-outs and copy-on-write.

Unfortunately, the creator ceased development on IFS because it became much too complex.

2.3 UnionFS

UnionFS is a unification file system designed to virtually unify different input file system while preserving UNIX semantics. It is able to combine an arbitrary number of branches by assigning each branch a priority, and resolve naming conflicts by the use of these priority assignments.

UnionFS was designed to address four important problems in file system unification: one, if the unified file system contains files and/or directories with the same name, it is unclear which one should be kept. Two, it is unclear how the file system should behave when one of the duplicate files are removed; it would be confounding to the user if the lower-priority file/directory were suddenly to appear after the higher priority file/directory is removed. Three, when read-only and read-write directories are mixed in the union, modifying arbitrary files will be problematic; if the user wishes to modify a file in a read-only directory, the file would have to be copied to a read-write capable and higher priority directory to be modified. The final problem is with cache coherency. The unionized file system should allow additions and removals from the unified virtual file system. Arbitrary insertions/deletions may cause incoherency of the directory name-lookup cache. We now examine how UnionFS has solved these problems.

To be as useful to as wide an audience as possible, UnionFS places very few restrictions on the number and types of branches allowed. Each branch can be read-write or read-only, and can be of any file system type. Dynamic insertion and removal of branchs are also allowed. The only restriction is that the top priority branch must be read-write. Many operations in UnionFS may be performed on multiple branches and file systems, so in order to adhere to UNIX semantics, they are designed to be atomic. If any part of an operation fails, it will fail as a unit.

The structure of UnionFS can be thought of as a fan-shape, with the leftmost branches having the highest priority; thus, as stated above, the leftmost branch is read-write, while the other branches can be of any type. It is very easy in this scheme to resolve naming conflicts. If there are multiple files/directories with the same absolute pathname, then the file/directory that is the leftmost in the fan (and thus has the highest priority) will be the one presented to the user.

If the user wishes to modify a file that resides within a read-write file system, then UnionFS simply allows the user to do so. If, however, the user wishes to modify a file that is within a read-only file system, the UnionFS has to complete an operation called a *copyup*, which is, in essence, a copy-on-write as described previously. In this operation, UnionFS essentially copies the file to be modified into a higher-prioirity branch that is also read-write (which is why the highest-priority must be read-write). During the copyup procedure, UnionFS may create an entire directory structure. When the copy is complete, the user is allowed to modify the file. Any modifications will be seen by the user because the newly modified file will mask the previous unmodified one.

When the user wishes to remove a file within a read-only branch, UnionFS creates a *whiteout*, similar to TFS, in a higher priority branch. Essentially, a whiteout in UnionFS is a zero-length file that lets the file system know that the whited out file should be treated as if it does not exist. A whiteout is named as .wh.F, where F is the name of the file/directory to be hidden. For example, a .wh.file1 tells the file system to not show file1 to the user, even if it exists.

2.4 AUFS

AUFS began as a modification of UnionFS (AUFS stands for Another UnionFS), but quickly grew into an independent project. It makes a number of improvements over UnionFS.

AUFS largely follows the same design as UnionFS, and improves upon its reliability and performance. We will not discuss the specifics of AUFS's implementation, but some of the notable improvements include: using an external inode number table to assign inode numbers, keeping the timestamps of file/directory during a copyup, and a seekable directory that supports NFS readdir.

2.5 Union Mounts

Union Mounts are another attempt at file system unification, and work as follows: the user first mounts a single file system, and then mounts another file system on top of the first. If there are any conflicts, the files/directories within the file system that is mounted later is given priority.

Traditionally, when a file system is mounted on top of a directory, everything within the directory is masked, and the user is given zero access to items within the directory until the masking file system is unmounted. This happens because within the kernel, the file systems are organized in the order that they are mounted. The file system that is mounted first is at the bottom of the stack, and grows from there. Only the files and directories within the top mounted file system is shown to the user.

With Union Mounts, the user is able to access information on all mounted file systems, as long as they are not masked by a later-mounted file system. Union Mounts, unlike other implementations such as UnionFS, do all of the directory merging within the VFS layer instead of creating an entirely new file system. The advantages of this are that the system is very simple and lightweight, and does not need to keep external metadata (such as whiteouts).

Currently, Union Mounts do not support writing to the unified file system as there are many implementation issues that need to be solved. Even so, it is a worthy player in the realm of file system unification.

2.6 Relation to LatticeFS

In building LatticeFS, we've adopted a number of innovative ideas from other union file systems. First, we adopted the fan-out branches scheme, but we do not modify any of our input branches. Instead, LatticeFS utilizes a store directory that is used when any file modification is required. Second, we also adopted the copyup idea during file modification, and finally, we utilize the white-out as a way to hide lower priority files and directories.

Chapter 3

Goals and Challenges

In this chapter, we outline the design goals and identify some of the challenges that were identified during the development of LatticeFS.

3.1 Goals

The initial motivation for creating LatticeFS was to solve the problem of virtual machine images. Suppose we begin with a virtual machine image A. From here, we modify A by adding/removing files and directories, which results in B (a modified image of A). Then, we modify A in a similar manner, but with different additions/removals, this time resulting in C (another modified image of A). From here, we'd like to be able to combine B and C into a new virtual machine image D such that it reflects all the changes that were made in both B and C. See 3-1.

Realizing that a virtual machine image is essentially a file system, we decided to design a filesystem unification tool that builds on existing system, and allow for novel ways for conflict resolution. We knew that while there were many file system unification tools in existence, none of them offered conflict resolution that was more clever than a simple priority queue. Additionally, existing tools did not handle special files any differently than normal files; files such as tarballs and virtual machine images are essentially directories in themselves, and could be conflict resolved in the same way. We therefore decided upon two main goals of LatticeFS:

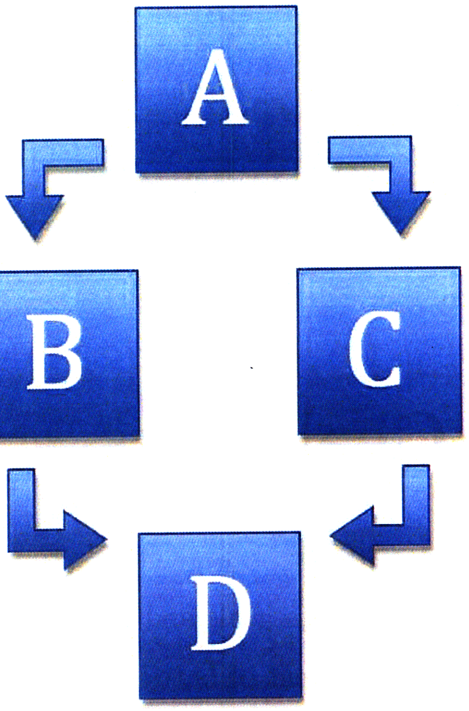

Figure 3-1: The lattice structure.

1. Design a system that would allow for conflict resolution techniques other than using priority assignments.

2. Design a system that is able to perform conflict resolution on file types that represent directories (such as tarballs and VM images).

When designing LatticeFS, we wanted the system to be able to support many different conflict resolution techniques and special file types; thus, we wanted the system to be as modular as possible. A new conflict resolution technique could be easily written and simply "plugged in". Additionally, the interface between the conflict resolution code and the rest of LatticeFS must be very simple so that a new technique could be written very easily. We have the same requirements of modularity and simplicity for the special file handlers.

Finally, we required that LatticeFS have two of the primary features of UnionFS:

1. LatticeFS must support an arbitrary number of input file systems.

2. LatticeFS must allow a branch to be read-only or read-write.

A few of the features of UnionFS (UNIX semantics, dynamic insertion/removal of branches) that we would have liked in LatticeFS proved be very difficult to implement in our design; we address this in the following section.

3.2 Challenges

One primary challenge lies in fulfilling the goal of sophisticated conflict resolution techniques. Going back to our original example of the virtual machines A, B, C, and D: suppose now that we have a conflict resolution scheme that will allow us to create virtual machine image D from B and C; now suppose that certain programs in image B and C had dependencies that conflicted. Clearly, there is no correct solution here: whatever LatticeFS chooses, at least one of the programs will break. In other words, it is extremely difficult to implement a conflict resolution scheme that will always "do the right thing". We therefore decided that, in general, the user knows best, and thus we designed the system to allow the user to choose file/directory that he/she deems correct. Additionally, we designed the system to be very modular, so if a resolution algorithm that always does the right thing is indeed possible, we may simply plug it in to our system without trouble.

The next challenge lies in the fulfillment of the goal of the special file handler. Archive files such as .tar and .zip files are fairly easy to handle, as they are simply represented as a directory. Additionally, there are modules written for most programming languages that allow easy interface with archive files. Virtual machine images, however, are much more difficult. Due to the scarcity of modules that interface with VM images, we may have to write our own interface, which would require copious amounts of difficult low-level system programming. Again, our system's modularity would allow us to "plug in" a solution in case we are unable to implement this in the relevant timeframe.

Finally, we identify two important features present in UnionFS that are very difficult to implement in LatticeFS. The first is adherence to UNIX semantics, specifically, atomicity. The second is the dynamic insertion and removal of branches. Atomicity is very difficult to uphold due to the fact that many LatticeFS system calls will require multiple underlying file system calls. We need to somehow group these calls in such a way that one failure among the group will reverse the effects of any previously completed calls. Finally, LatticeFS cannot support the dynamic addition/removal of branches because it is built under FUSE, which is missing such a feature.

Chapter 4

Design

In this chapter, we give an overview of LatticeFS's design considerations.

4.1 VFS

VFS, or the Virtual Filesystem Switch, is a subsystem built into all Unix-based operating systems; it is the reason that Unix is able to easily coexist with other operating systems' file formats, and it is the reason that we are able to build LatticeFS.

The idea behind VFS is that, during each file system call (read, write, etc), the kernel substitutes in the system call pertaining to the file system in questions, and is thus able to interface with a wide variety of different file system types. In other words, it is an abstraction layer between the high level system calls and the low level hardware interfaces.

For example, suppose we wish to copy a file from a directory that is on an NTFS disk to another directory that is on an EXT3 disk. The copy, or cp, system call would then be translated by the VFS layer, and the corresponding NTFS "version" of the system call will be invoked. The results are correspondingly passed on to the EXT3 directory. See 4-1.

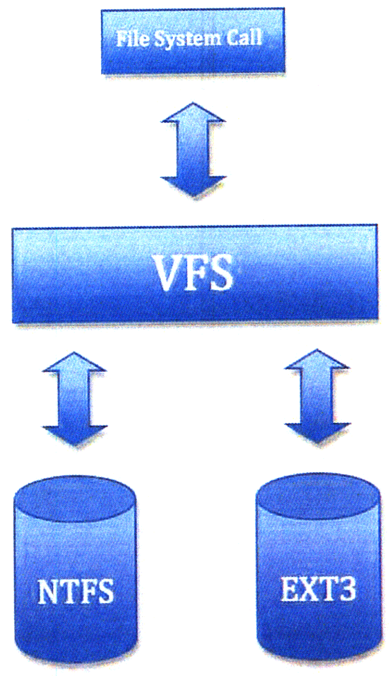

Figure 4-1: The VFS layer.

4.2 Fuse

LatticeFS is built on fuse (Filesystem in UserSpacE). Fuse is essentially a loadable kernel module that allows non-privileged users to create their own file systems. Fuse allows us to easily create our own file systems without having to work with the VFS layer ourselves; all of the interface between user-level code and the VFS are taken care of by fuse.

A transaction between the fuse module and the VFS layer works as follows: first, our custom file system residing in userspace makes a call to the fuse library, which in turn calls the fuse module residing in kernel space. This module in turn interacts with VFS. A simple diagram of this interaction can be seen in 4-2.

4.3 Two Versions of LatticeFS

Initially, we designed a version of LatticeFS that handles conflict resolution at run-time; we later wrote another version of LatticeFS for comparison; this time, we pre-

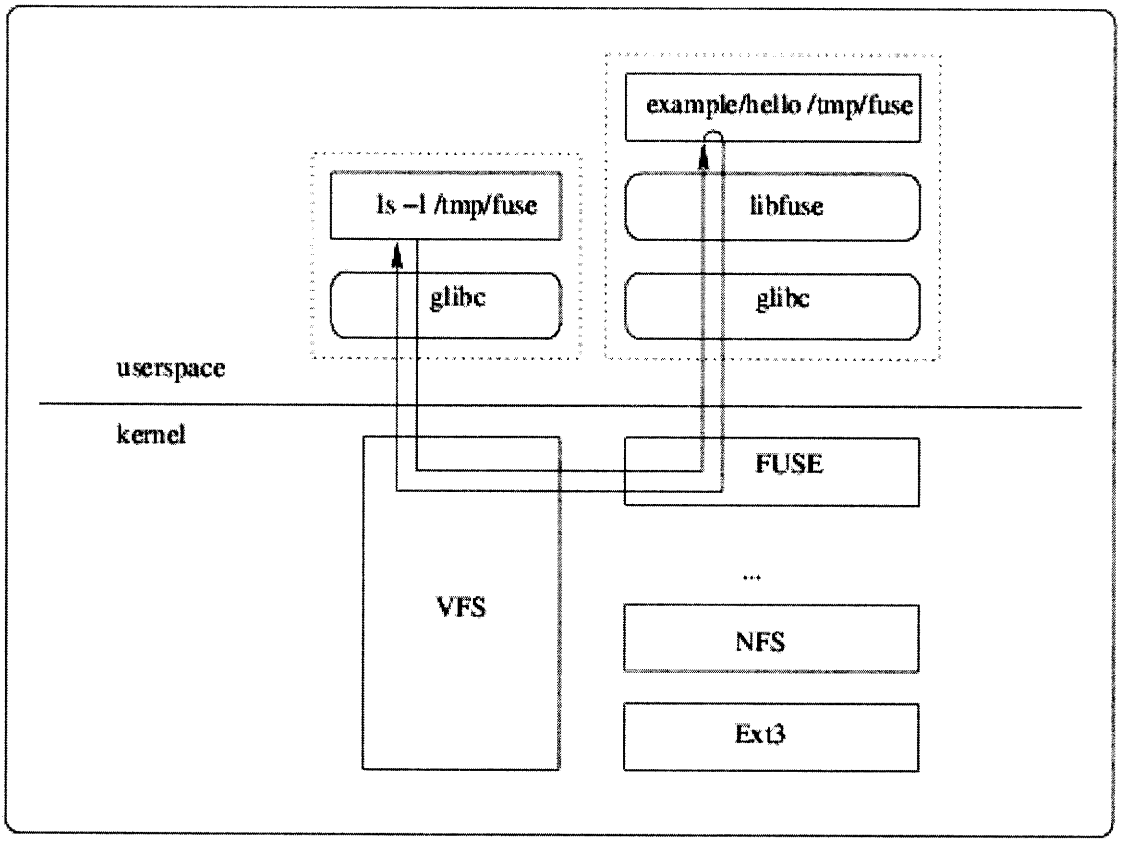

Figure 4-2: The fuse module. [6]

compute all potential conflicts and resolve them before runtime. In the following sections, we compare and contrast the designs of the two.

4.3.1 Storage

In the previous chapter, we stated that LatticeFS assumes that all input file systems are read-only, but nonetheless supports read-write capabilities on the newly created virtual file system. To accomplish this task with the runtime version of LatticeFS, we decided to enlist the use of a "store" directory. If the user wishes to modify a file/directory residing within a read-only file system, LatticeFS first copies that file/directory to the store directory, and then allows the user to edit the copied version.

We considered many different structures for the store directory. One potential implementation was to store all files within the root of the store directory. Each file would be named by its absolute pathname, so a file copied from /dir1/file1 would actually have the string "/dir1/file1" as its name. This implementation would be very simple; there would be very little computation overhead, and the implementation of the required fuse system calls would be easy. However, we would not be able to support directories, and there are serious scaling issues with placing a very large number of files into a single directory. A frequently used file system could easily have more than ten thousand edited files, which would slow this implementation to a crawl.

In the end, we settled upon a design that we felt optimized the difficulty of implementation versus its scalability and performance. In this implementation, we require that the store directory mirror the structure of the input file systems. In order to simplify the implementation of most of the required fuse file system calls, we lazily maintain the required structure. In other words, when we need to copy a file or directory over to store for editing, we create the underlying directory structure as well (but none of the files). For example, suppose that at some point in time, the store directory is empty, and that we wish to edit the file /dir1/dir2/file1. We would first create /dir1 within store, then /dir1/dir2, then we finally copy over /dir1/dir2/file1. If, say, there exists a /dir1/file1, we would NOT copy this file over to store.

In fact, the runtime version LatticeFS's structure is very similar to that of UnionFS.

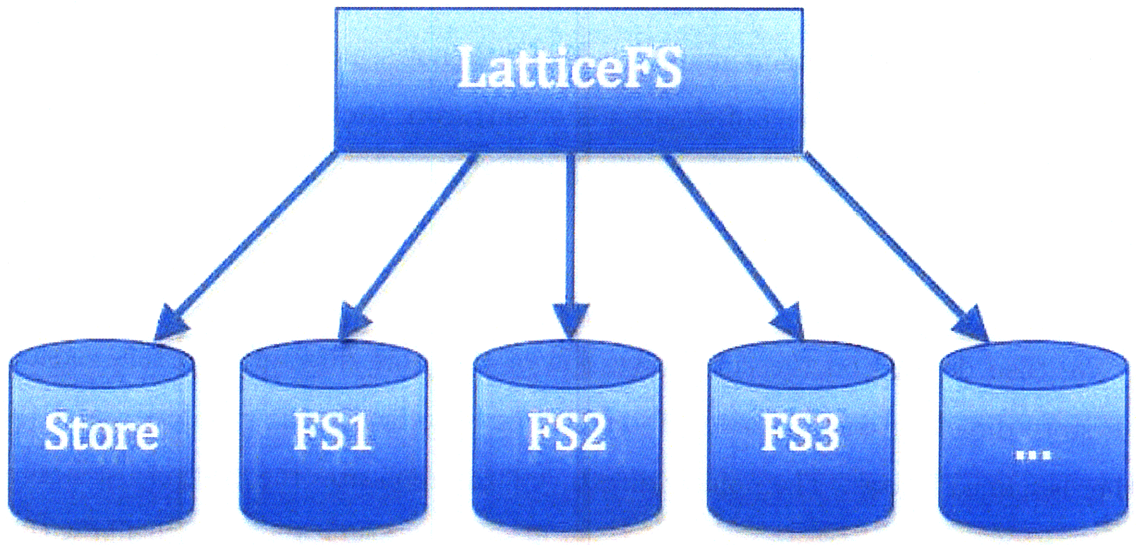

Figure 4-3: The Fan-out Structure.

We also adopt a fan-out shape for our input file systems; the difference is that we treat all input file systems as read-only, and we add the store directory into the left-most position as the highest-priority file system. This way, when a file is copied over for editing, the changes would be seen by the user because the store directory is of a higher priority than all input file systems. See 4-3.

The same read-write requirement holds for the pre-compiled version of LatticeFS. For the pre-compiled version of LatticeFS, we also considered many different designs for the store directory structure. The first idea was that we use a hash table data structure where the key is the absolute pathname of a file/directory, and the value attached to that key is the actual location of the item that we're looking for. We would use this hash table along with a storage directory. For example, suppose that we create a Lattice file system from the directory /A. LatticeFS would then create a storage directory called, say, /A-store. Initially, we create a single entry in the hash table: (/, /A). This designates that all files within the root directory are to be found in /A, which is exactly what we want. If we then modify a file within /A, say /A/file1, we would copy /A/file1 into /A-store/file1, and update the hash table with the entry (file1, /A-store). This designates that file1 is located within /A-store.

The preceding idea is fairly simple to implement, and conflict resolution can also be implemented easily. Unfortunately, it also does not scale very well. A frequently used system could have tens of thousands of entries in the hash table, which would make memory requirements skyrocket, as well as the runtime of many required system calls.

We then decided on the current version of the store directory. In this version, the store directory contains two subdirectories: one is named "pointers", and the other is named "data". The pointer directory lazily replicates the directory structure of the mounted file systems, and contains files of a specific type. Each file within the pointer directory can be thought of as a "pointer", but instead of containing the memory addresses of objects as they normally would, our pointers contain the absolute pathname of the actual location of a file/directory. For example, suppose a file called "file1" exists in the pointer directory; further suppose that file1 contains the string "/dir1/file1". This tells LatticeFS that the actual file "file1" is located at /dir1/file1.

The previously described structure has many benefits: first, there is no need to implement a white-out system because we can represent the removal of a file/directory by simply removing the pointer from the pointer directory. Second, conflict resolution will be fairly straightforward to implement because we can simply combine the two store directories' pointer directories. Third, the structure allows us to implement an important optimization: when the file system is first created, no files will have been edited, and all files will be exactly as they are on the input file systems. At this point, we can simply create pointers for the files/directories in the root directory. When the user wishes to edit a file deeper in the directory tree, we simply replicate the directory structure up to that file within the pointer and data directories, copy the file into the data directory, and finally create a pointer file that points to the just-copied file within the data directory. This system will be covered in greater detail in the following chapter.

4.3.2 Conflict Resolution

One of the primary goals of the Lattice file system is to explore more interesting ways of doing namespace conflict resolution. Most current namespace unification systems, including Union FS, use a simple priority queue to determine which of a set of duplicate files would be exposed to the user. The first iteration of Lattice FS did use such a system, in which priority was given to one of the two source file systems; in the beginning, this was simply hard-coded into the Fuse implementation. For the next iteration of Lattice FS, we decided to modularize the conflict resolution code by encapsulating it in a class called Resolver. The constructor for Resolver takes in a Policy class, which determines how each conflict is resolved. The Policy object is responsible for determining the absolute path of the file to be exposed to the user; thus, if we wished to implement a different conflict resolution policy, we can simply create another Policy object.

For the compile-time version of LatticeFS, we also explored the realm of resolving special files whose internal structure was essentially that of a directory tree. Such files include archives (zip or tar) and virtual disk images. In our implementation of LatticeFS, we are able to support the resolution of .tar files. The primary goal for the design of this system is modularity; it should be very simple to write a plug-in that interfaces with LatticeFS, and is able to "decode" a special file type into a simple directory structure to be resolved. The special file handler is called by the pre-compilation conflict resolution tool.

The runtime version of LatticeFS creates a Resolver during initialization, and calls on it upon reaching a conflict. The compile-time version of LatticeFS calls upon the resolver during the pre-compilation conflict-resolution phase.

4.3.3 Comparison

In this section, we compare and contrast the two implementations of LatticeFS, and attempt to assess the pros and cons of each.

The runtime version of LatticeFS will be slower in most cases due to the fact that it has to resolve conflicts during runtime. However, it is also easier to use because there is no need to run pre-compilation code to merge store directories. Additionally, unlike the compile-time version, it does not require that, given multiple inputs, each input must be a Lattice file system, thereby expediting the process of setting up a multiple-input virtual file system. The compile-time version is also more "fragile". It is required that no changes be made to the store directories once they are mounted; otherwise, the file system could be internally inconsistent, and errors will undoubtedly occur.

As stated above, the compile-time version of LatticeFS is able to resolve special file types with directory tree-like internal structures. Unfortunately, this did not seem extensible enough in the runtime version of LatticeFS; for some filetypes, the amount of computation required could be significant, which would simply not be permissible at run-time. For example, if there are two conflicting tar archives that are very large, there would be a significant delay during which the combined tar archive is being created during runtime; the user experience would therefore be negatively impacted.

Chapter 5

Implementation

In this chapter, we explore in detail the inner workings of LatticeFS in its two implementations: the version with runtime conflict resolution and the version with pre-computed conflict resolution.

5.1 Code Structure

As stated in the previous chapter, we chose to implement LatticeFS using Fuse. Fuse file systems can be written in any language that has the required bindings to the Fuse libraries. Python is the language of choice for our needs; it is easy to work with, good for prototyping, and boasts very good Fuse bindings. To implement a Fuse file system, we simply have to implement a number of required methods. We will look at these implementations in greater detail in a later section.

The code structure of the runtime version of LatticeFS is fairly simple. The main class is called LatticeBindings, and implements the Fuse object. Within LatticeBindings, we define the Fuse file system's required methods. Each method calls the classes that handle conflict resolution and white-out handling as required. The conflict resolution system is encapsulated in two classes: the Resolver class and the Policy class. The Resolver class serves as a simple wrapper that takes as input a conflict resolution policy (as represented by the Policy class), and outputs the result of conflict resolution. The whiteout system is implemented as the WhiteOut class. It

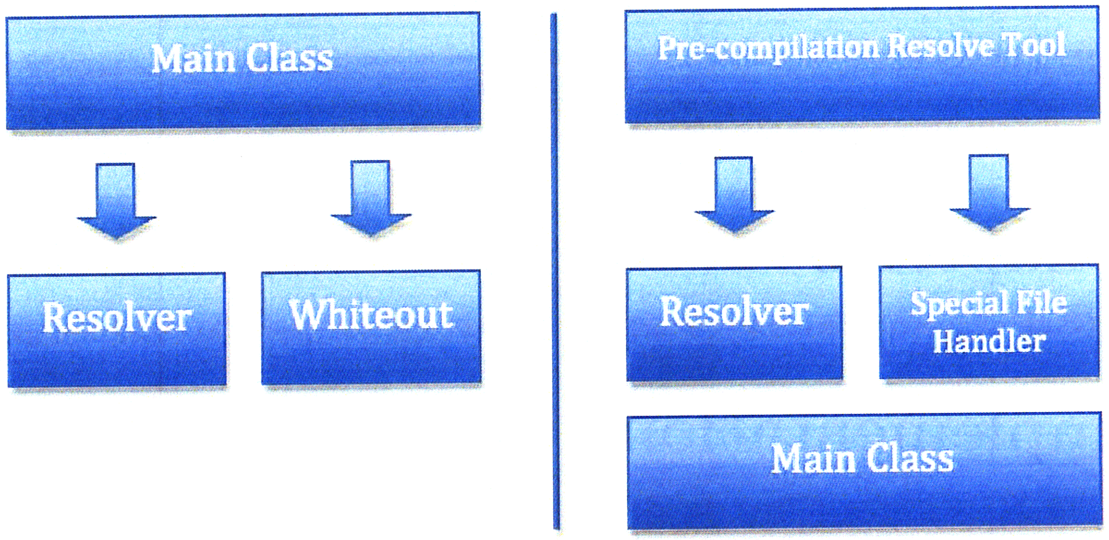

Figure 5-1: The code structure of the two types of LatticeFS; the runtime version is on the left, and the compile-time version is on the right.

has methods to create and remove whiteout, and to change the absolute pathname of the file/directory in question.

The code structure of the compile-time version of LatticeFS is slightly more complicated. In this version, the LatticeBindings class still implements Fuse, but does not interact with any other class due to the fact that all conflict resolution has been handled by runtime. We define a new class called pcresolvetool to be run before mounting LatticeFS, the purpose of which is to resolve any potential conflicts. The pcresolvetool in turn calls two classes: the Resolver class, and the special file handler class. The resolver class behaves as before, while the special file handler class takes as input two absolute pathnames, and determines whether they represent special files that LatticeFS can handle; if so, then the file handler attempts to resolve them. See 5-1.

5.2 Whiteout Implementation

LatticeFS assumes that all input file systems are read-only, so it is prohibited from removing files and directories. As stated in the previous chapter, we solve this problem through the use of white-outs.

Within Union FS, a whiteout is represented as a zero-length file that resides within the same directory as the file to be whited out. The file name takes the form .wh.F, where F is the name of the file to be whited out. For the runtime version of Lattice FS, we originally implemented the whiteout using the same method, but decided that this method would effectively restrict the users ability to name arbitrary files, since all file names of the form .wh.* are taken. We then decided to move all whiteout files into a directory of their own. The directory structure would reflect exactly that of the unified file system, and would contain only whiteout files; this implementation did solve the namespace restriction problem, but was not very elegant. Additionally, it introduces the overhead of having to maintain the exact directory structure of the underlying file system. Finally, we decided upon the current implementation: essentially, we keep a dictionary of all whiteouts. Each time a whiteout is required, the absolute pathname to the file to be erased is written into the dictionary as a key. The value associated with the key is 0. Thus, it is a simple matter to determine if a file has an associated whiteout: we simply check for existence of a key equal to the file pathname. When a whiteout is added/removed, we propagate the in-memory dictionary into a file on disk through the use of Python's Pickle package, which allows us to write arbitrary in-memory objects to disk. The file is reloaded when the system is re-mounted.

The WhiteOut object has five methods: the constructor, set_path, has_whiteout, make_whiteout, and remove_whiteout. The constructor takes as input the location of the pickled dictionary, and, if required, initializes a new dictionary. The set_path method takes as input the pathname of a file/directory, and sets the internal stored pathname to be the input. The has_whiteout, make_whiteout, and remove_whiteout methods all work with the internally stored path set by set_path. Has_whiteout simply checks for existence of the pathname within the dictionary. Make_whiteout inserts a new entry into the dictionary; the entry has the pathname as the key, and zero as the value. Remove_whiteout will delete the entry whose key is the pathname, and then propagate the altered dictionary to disk using pickle.

5.3 Conflict Resolution

Conflict resolution is handled by two classes: the Resolver and the Policy. The Policy class embodies the different conflict resolution strategies that we can use, while the Resolver class is essentially a wrapper for the different policies.

The Policy class defines three methods: the constructor, set_path, and resolve. The constructor takes as input the pathnames of the input file systems, the pathnames of the two files/directories in conflict, and the whiteout object that we have previously discussed. The set_path method sets the internally maintained pathname to be the input. The resolve method returns the pathname of the file/directory that should take priority. We designed the conflict resolution system to be very modular; thus, if one wishes to implement a new Policy for conflict resolution, one simply has to write a new class that overloads the resolve method of the Policy base class. We have written a sample Policy named PriorityPolicy. Essentially, it reads the set of input file systems as a priority queue, and returns the file/directory located in the file system closest to the front of the list; this is the UnionFS policy. We have also written a Policy that queries the user whenever a conflict is met; this policy would not work very well in the runtime version of LatticeFS, so we have implemented it only for the compile-time version.

The Resolver wrapper class contains three methods that are identical to that of the Policy class: the constructor, set_path, and resolve. The constructor takes as input a Policy object, and the other two methods simply invoke the corresponding methods of the internally kept Policy object.

For the compile-time version of LatticeFS, we added the conflict resolution tool that is to be run before mounting our file system. The resolution tool consists of a main class and a helper function called combine_db. The main function takes as input a set of directories that represent the file systems to be merged. If the number of of inputs is one, then it is assumed that the one directory is not, in fact, a LatticeFS, so a store directory will be created for it. The newly created store directory will be the input for the main LatticeFS class, and the resulting mounted file system will mirror exactly the file system that the store directory was created from.

If the number of inputs to the resolution tool is more than one, then the helper function combine_db is called. The following pseudocode describes the combine_db function's operation:

```
COMBINE_DB(sources)
 1  Recursively copy directory tree of sources[0] into new_store
 2  for source∈sources[1:]
 3      do outstanding ←all items in source's root directory
 4          while outstanding is not empty
 5              do c ← outstanding.pop()
 6                  if c is a file
 7                    then if there is already something with c's pathname
 8                            then call special file handler to see if it applies
 9                                 call resolver to see which file/directory we keep
10                            else copy c into new_store
11                    else  if there is already something with c's pathname
12                            then call resolver to see which file/directory we keep
13                                 if we keep c
14                                   then outstanding←outstanding + all items in c
15                            else copy c into new_store
16                                 outstanding←outstanding + all items in c
```

The combine_db method is fairly simple: we take the first input source, and copy its entire directory tree (line 1) to our new store directory. We then crawl breadthfirst through each other input source, and at each file/directory, determine whether it conflicts with another file/directory that is already in the new store directory. If there is indeed a conflict, we check to see if the special file handler applies; if not, then we call on the resolver to determine which of the files/directories will be copied to the new store directory.

The compile-time version of LatticeFS also boasts the special file handler. Currently, we have implemented the handler for tar archives using Python's tarfile module. Since archive files have essentially the same tree structure as that of file systems, we use the same idea to create a new .tar file from two conflicting .tar files.

5.4 Runtime Implementation

In the following sections, we discuss implementation details of a number of representative methods within the main class: LatticeBindings. We first examine the implementation of the runtime version.

5.4.1 getattr

The getattr method is a good example of the layout of most of the methods in LatticeBindings. First, we create a Policy object. Then, we create a Resolver object, with input as the Policy that we have just created. We then use the resolver to determine the correct pathname. We then create a fuse.Stat object from this pathname and return it to the user.

5.4.2 readdir

The readdir system call has the task of compiling the set of all items contained within the given directory. The following pseudocode describes its operation:

```
READDIR($path$)
1  $policy$ ←new policy object
```

```
2 resolver ←new resolver object
3 paths ←entries in path, for each root location (the input file systems and store)
4 entries ← []
5 for p∈paths
6     do if p should be show according to resolver
7           then entries.add(p)
8 for entry∈entries
9     do yield fuse.Direntry(entry)
```

The readdir method first gathers the set of all potential entries to be returned. It does this by looking through all directories with the given input pathname, located in any of the input file systems, or in store. All of these are then checked by the resolver; we then create fuse.Direntry objects for each file/directory that the resolver determines.

5.4.3 rename

The rename system call is arguably the most difficult and complicated to write. It takes in a source and destination, and should rename the file/directory at source to that of destination. We have available to us the Python os module, which includes a rename system call. We need to determine, however, the location of the source; if it resides within one of the read-only input file systems, then the situation becomes much more compilcated. The following pseudocode describes its operation:

```
RENAME(source, destination)
1 policy ←new policy object
2 resolver ←new resolver object
3 resolved_path ← resolver.resolve()
4 if resolved_path is in store
5    then call os.rename on source and destination
6    else  for s ∈source file systems
```

```
 7                do if resolved_path is in s
 8                    then create the underlying directory structure if it does not exist
 9                        copy over the file/directory in question to the store directory
10                        call os.rename on the newly copied file/directory
```

The rename method begins in the same way as the other system calls: the creation of a Policy and Resolver. It then calls Resolver's resolve() method, thereby determining which file system the file/directory to be renamed is located. If the location is our store directory (which is read-write), then we can simply call the os.rename() method, and we're done. If, however, the location is one of the read-only input file systems, then we have to create the necessary directory structure in store (remember we do this only when we need to), and then copy the file/directory over. When this is done, we can simply call rename on the newly copied over object.

5.4.4 unlink

The unlink call takes as input a pathname, and attempts to remove the corresponding file/directory from the file system. The following is pseudocode for the unlink call:

```
UNLINK(path)
 1  policy ← new policy object
 2  resolver ← new resolver object
 3  resolved_path ← resolver.resolve()
 4  if resolved_path is in store
 5     then call os.unlink on path
 6          for s ∈ source file systems
 7              do if path exists in s
 8                    then create a whiteout for path
 9     else for s ∈ source file systems
10              do if resolved_path is in s
11                    then create a whiteout for path
```

Again, unlink begins in the same way as before: the creation of Resolver and Policy. We then check the resolved path (which is the return value of the resolver) against the store directory and the source file systems. If resolved path is in store, then we call os.unlink() on path. We then have to check if path occurs in any of the source file systems; if so, then we have to create a whiteout to hide them. If resolved path points to a source file system, then we have to make a whiteout for it (remember that source file systems are all read-only, so we cannot actually unlink anything).

5.5 Compile-time Implementation

We now discuss the implementation of the above system calls within the compile-time version of LatticeFS. In general, the system call implementations are more complex.

5.5.1 getattr

The getattr method, again, is a good example of the layout of most of the methods in LatticeBindings. In this version, all of the conflict resolution is complete, so we do not instantiate any Policy or Resolver objects. The following is pseudocode for the getattr method:

```
GETATTR(path)
1  components ←tokenize path and return a deque
2  pointer_path ←path to our store's pointer directory
3  while components is not empty
4      do if pointer_path is not a directory
5             then break
6             else  pointer_path ←pointer_path concatenated with components.pop()
7  real_path ←contents of file at pointer_path
8  while components is not empty
9      do real_path ←real_path concatenated with components.pop()
```

```
10  create fuse.Stat object from real_path and return
```

The getattr method includes a subroutine that we will see again and again: we tokenize the input full pathname into a set of directory names, and progressively add each "piece" to our pointer path until we determine that the resulting pathname is no longer a directory. This is because the pointer directory holds "pointer" files whose contents are the actual locations of the files/directories that we are looking for. These pointer files can occur at any point in our pathname, so we must check them all. Once the pointer file is found, we read it to determine the actual path, and create a Stat object from it.

5.5.2 readdir

The compile-time implementation of readdir contains the same subroutine described above for getattr. The following details its operation:

```
READDIR(path)
1   components ←tokenize path and return a deque
2   pointer_path ←path to our store's pointer directory
3   while components is not empty
4       do if pointer_path is not a directory
5             then break
6             else  pointer_path ←pointer_path concatenated with components.pop()
7   real_path ←contents of file at pointer_path
8   while components is not empty
9       do real_path ←real_path concatenated with components.pop()
10  entries ←contents of directory at real_pathfor entry∈entries
11      do yield fuse.Direntry(entry)
```

The readdir method runs the subroutine to determine the real location of our directory in question; we then create fuse.Direntry objects for each file/directory that resides within this real location.

5.5.3 rename

The following is pseudocode for rename:

```
RENAME(source, destination)
components ←tokenize source and return a deque
pointer_path ←path to our store's pointer directory
while components is not empty
    do if pointer_path is not a directory
          then break
          else pointer_path ←pointer_path concatenated with components.pop()
real_path ←contents of file at pointer_path
while components is not empty
    do real_path ←real_path concatenated with components.pop()
components ←tokenize source and return a deque
temp_path ← ""
while components has more than one item
    do temp_path ←temp_path concatenated with components.pop()
       if temp_path is a file
          then actual_path ←contents of file at temp_path
               unlink temp_path
               make a directory at temp_path
               for item ∈directory at actual_path
                   do $item_path$ ←concatenation of actual_path and item
                      write item_path into contents of corresponding pointer file
call os.rename on the data directory's source and destination
call os.rename on the pointer directory's source and destination
```

The rename call begins as before, with the subroutine to determine the real location of the input pathname. It then creates the directory structure within the store directory, and updates the pointer files' contents to reflect the new location. Finally, it calls os.rename to move the files/directories to be renamed to their correct positions.

5.5.4 unlink

The unlink call operates as detailed by the following:

```
UNLINK(path)
 1  components ←tokenize path and return a deque
 2  pointer_path ←path to our store's pointer directory
 3  while components is not empty
 4      do if pointer_path is not a directory
 5            then break
 6            else  pointer_path ←pointer_path concatenated with components.pop()
 7  real_path ←contents of file at pointer_path
 8  while components is not empty
 9      do real_path ←real_path concatenated with components.pop()
10  components ←tokenize source and return a deque
11  temp_path ← ""
12  while components has more than one item
13      do temp_path ←temp_path concatenated with components.pop()
14         if temp_path is a file
15            then actual_path ←contents of file at temp_path
16                 unlink temp_path
17                 make a directory at temp_path
18                 for item ∈directory at actual_path
19                     do $item_path$ ←concatenation of actual_path and item
```

```
                write item_path into contents of corresponding pointer file
call os.unlink on the data directory's path
call os.unlink on the pointer directory's path

```

Unlink begins with the subroutine that has been common to all of the previous system calls, and also utilizes the rename system call's subroutine of creating the necessary directory structure. It then calls unlink on the appropriate path within both the pointer and data directories.

Chapter 6

Performance Evaluation

In this chapter, we evaluate both versions of LatticeFS on a set of benchmarks that are designed to determine the performance cost of basic file access tasks. Since there is some overhead to running LatticeFS versus other more conventional file systems, we aim to determine whether the performance cost is prohibitive to everyday use. We first describe the testing methodology, and then present the results.

6.1 Testing Methodology

We have written a set of python scripts that perform basic file access tasks such as open, close, read, and write. Each command was run one hundred times, and the loop was timed using python's time module. There are two dimensions to the test. In one dimension, we examine the performance of the two versions of LatticeFS as they compare to a conventional Unix ext3 file system. By this comparison, we aim to evaluate whether the overhead in running LatticeFS is prohibitively high.

In the second dimension, we run each set of file access tasks on three volumes: one is very small, and only has two directory levels; one is medium sized, with one more directory level, and twice the number of files and directories. Finally, one is the largest of the three, and has one more directory level, and twice the number of files and directories as the medium sized one. The file used for our testing is located in the directory that is the farthest down the directory tree. Here, we aim to evaluate whether the implementation of LatticeFS scales well, and is able to handle very large systems.

The machine that we use has a 3.0ghz pentium 4 processor with hyperthreading, one gigabyte of RAM, and a 100gb 7200 RPM IDE hard disk. It is running Ubuntu Linux version 8.04.

We have benchmarked five tasks:

1. Open a file for reading, and then close it (no reading is done).
2. Open a file for writing, and then close it (no writing is done).
3. Open a file for reading, read its contents, and then close it.
4. Open a file for writing, write about 100 bytes to it, and then close it.
5. Open a file for with the write and creation flags, write about 100 bytes to it, close it, and unlink it.

6.2 Results

Figure 6-1 summarizes our results: the numbers are the average amount of time taken with each task, in milliseconds. The read only, write only, and unlink only times are calculated by subtracting the read, write, and unlink times from the top two rows.

For benchmark 1 (open a file for reading, and then close it), we can see that both versions of LatticeFS are much slower than the regular ext3 file system. The compile-time version is about 50 times slower, while the runtime version is around 30 times slower (6-2). We can see a similar problem with benchmark 2 (6-3, open a file for writing, and then close it), in which the results are essentially the same. The reason for this is twofold: first, in order to open a file within LatticeFS, we must first create a LatticeFile object which supports the use of multiple input file systems; this introduces significant overhead to the action of opening a file. The second reason is that we have implemented LatticeFS using Fuse, and in Python. Python is fairly slow because it is an interpreted language, and Fuse simply adds

	Unix ext3 *small*	Unix ext3 *medium*	Unix ext3 *large*	compile-time *small*	compile-time *medium*	compile-time *large*	runtime *small*	runtime *medium*	runtime *large*
open for read	0.0104	0.0104	0.0103	0.5693	0.5442	0.5546	0.3057	0.3044	0.2990
open for write	0.0144	0.0140	0.0177	0.7019	0.7021	0.7038	0.4291	0.4419	0.4262
read	0.0211	0.0204	0.2084	0.7486	0.7598	0.7563	0.4865	0.4791	0.4761
write	0.0485	0.0486	0.0472	0.7623	0.7621	0.7767	0.4987	0.4843	0.5066
unlink	0.0672	0.0614	0.0636	1.0022	1.0033	1.0104	1.1267	1.1363	1.1151
read only	0.0107	0.0101	0.0198	0.1793	0.2156	0.2017	0.1808	0.1747	0.1771
write only	0.0341	0.0346	0.0295	0.0604	0.0600	0.0729	0.0696	0.0424	0.0804
unlink only	0.0187	0.0128	0.0164	0.2399	0.2413	0.2338	0.6281	0.6520	0.6085

Figure 6-1: Benchmark results. The last six columns are of LatticeFS

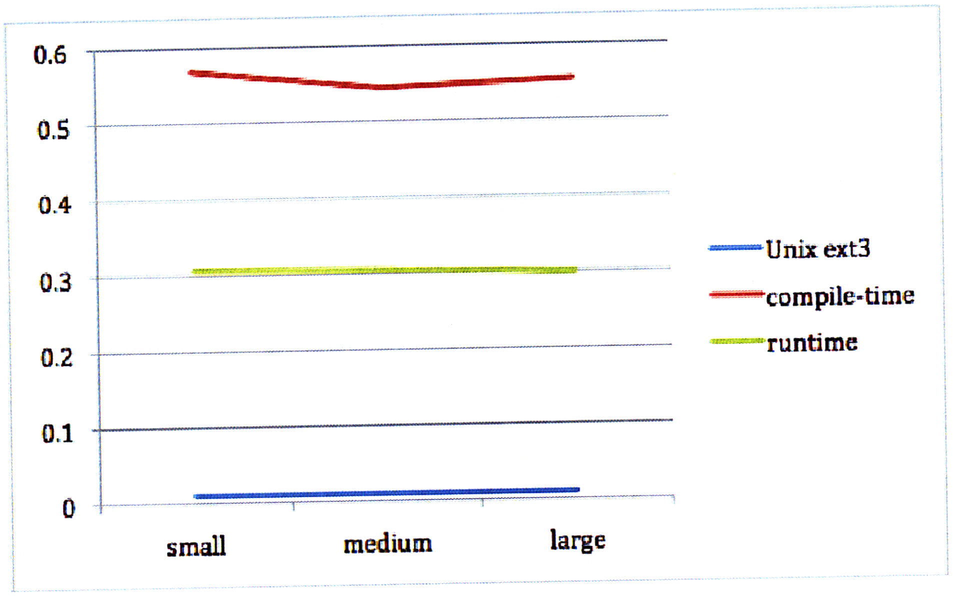

Figure 6-2: Benchmark 1: open for read

another layer of abstraction between our file system and hardware, which requires overhead to maintain.

For benchmark 3 (6-4, read a file), we test the speed of file reading. Here, LatticeFS fares much better than before, and is about 20 times slower for the compile-time version, and 17 times slower for the runtime version. The speed gain seems to be from the fact that there is no longer overhead in creating a LatticeFile object.

The results of benchmark 4 (6-5, write to a file) are rather surprising; here, both versions of LatticeFS are about half of the speed of ext3. This is a very good result considering the significant overhead in utilizing Fuse. It seems that it is using soft updates, and not all writes are being propagated immediately to disk; thus, we no longer have the overhead that we did before.

For benchmark 5 (6-6, unlink a file), the compile-time version of LatticeFS is around 20 times slower, while the runtime version is a massive 60 times slower. This is due to the difference in the unlink implementations. For the compile-time version, we simply unlink the pointer file, and we are done. For the runtime version, however, we must pinpoint its location (store or one of the input file systems), and then either create a whiteout or remove the file, depending on its location.

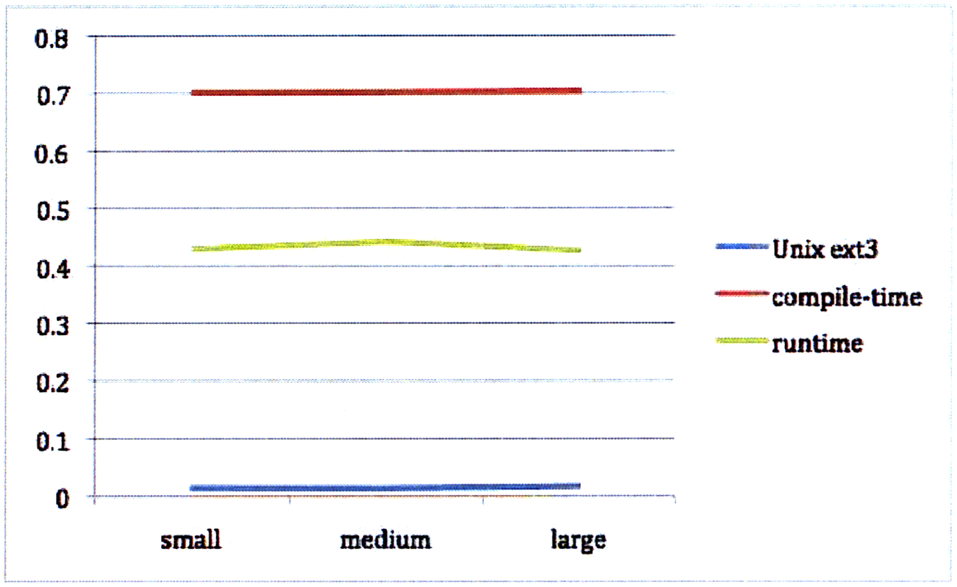

Figure 6-3: Benchmark 2: open for write

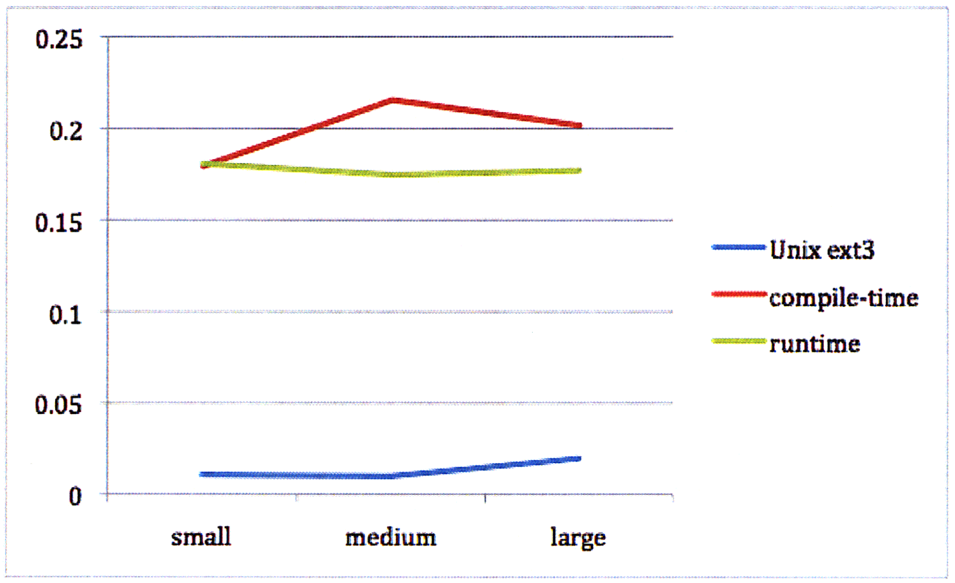

Figure 6-4: Benchmark 3: read a file

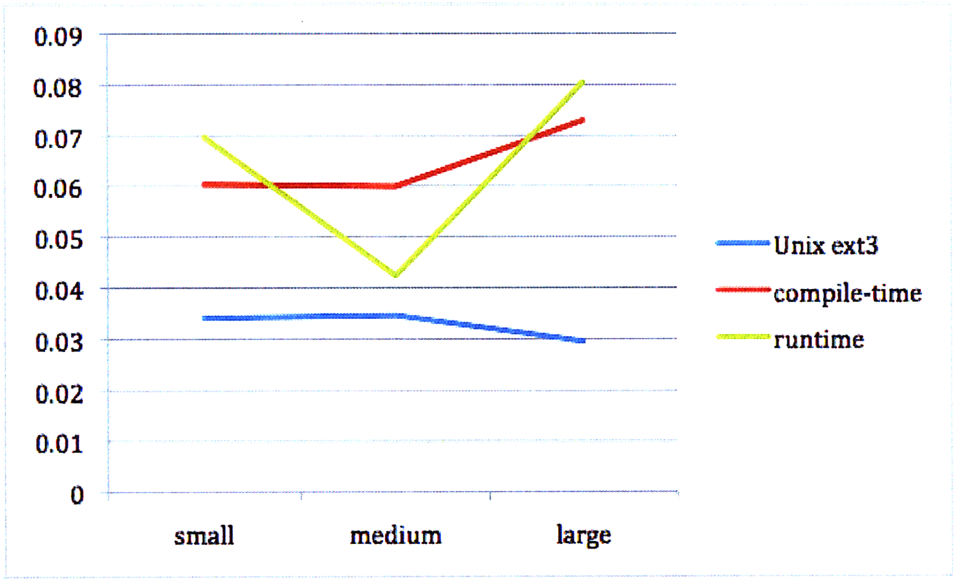

Figure 6-5: Benchmark 4: write to a file

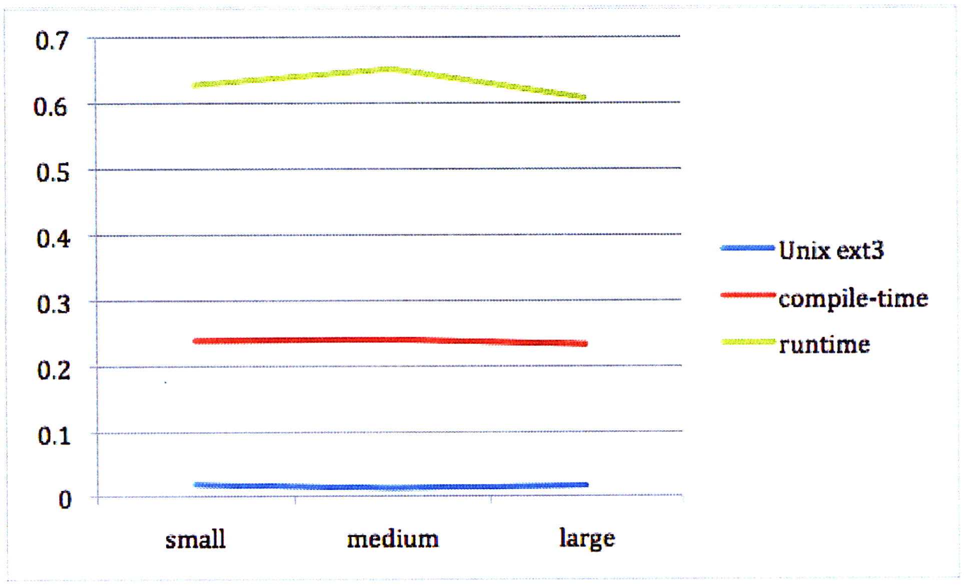

Figure 6-6: Benchmark 5: unlink a file

From the above results, we can gather that the Python and Fuse overhead makes operations around 20 times slower. Additional computation required for opening a file and unlinking brings the time required for those operations up to 60 times slower than a normal ext3 file system. Nevertheless, basic file operations are still less than 1 millisecond on either version of LatticeFS, and the system is acceptable for simple I/O tasks. For I/O intensive applications, however, the current implementations probably will not do. Python was chosen for its ease of use, and if we wished to create a significantly faster version of LatticeFS, we would have to re-implement it in a compiled language. Additionally, we can see that even though LatticeFS is slower than a conventional Unix file system, it scales fairly well. The differences in performance on the small, medium, and large directories are essentially non-existent; however, the fact that LatticeFS is about 20 times slower than a traditional Unix file system suggests that it will not do for large systems.

Chapter 7

Conclusion

7.1 Future Work

In order for LatticeFS to be usable as a fully functional file system, there is more work to be done. The nature of the LatticeFS project was to explore different aspects of file system unification and conflict resolution; thus, certain system calls that were not required were, in fact, not implemented (chmod, chown, utime). Implementing these functions would be the first step in making LatticeFS into a fully functional file system. Additionally, as can be seen in the previous chapter, LatticeFS is significantly slower than a conventional Unix file system, and while it is acceptable for basic I/O tasks, I/O intensive applications would have trouble on LatticeFS. To solve this problem, we can re-implement LatticeFS using a compiled language, thereby speeding it up significantly.

For a performance-oriented version of LatticeFS, a language such as C++ would be ideal. It is significantly faster than python, and is similar enough in syntax so a rewrite of the code would be fairly simple. As stated in the previous chapter, an additional cost in performance comes from the Fuse system itself. For our faster version of LatticeFS, we could directly interface with VFS instead of going through the Fuse module; this should eliminate significant overhead.

We have designed a very modular conflict resolution system for LatticeFS in which it is very easy to implement novel resolution algorithms. Currently, we have implemented some very simple ones, and developing more sophisticated algorithms will be a priority for the future. For example, utilizing the lattice structure of our file system, we can determine the case in which a file has been modified from its original state in one input file system, but was left alone in a different input file system. In this case, it would be logical to automatically keep the changed file without having to query the user to make a decision.

Finally, we have implemented a special file handler that is able to resolve certain file types that exhibit directory-tree structure. Currently, LatticeFS supports only the tar archive file; for the future, we aim to support many more file types.

7.2 Conclusion

In this thesis, we have examined our implementation of LatticeFS, a tool designed to merge an arbitrary number of input file systems into a single usable file system. File system unification is useful for a myriad of tasks from database maintenance to system updates. We have focused our attention on the conflict resolution aspect of file system unification.

Specifically, we have explored different strategies for conflict resolution and special file handling; LatticeFS is designed to be very modular with respect to conflict resolution strategies, and, in this regard, we believe that we have succeeded. As a proof of concept, we have implemented a few very simple resolution strategies, and support the handling of tar archive files. As shown by performance testing, while LatticeFS is slower than conventional Unix file systems, it is nonetheless usable for everyday tasks.

Bibliography

[1] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, Erez Zadok, and Mohammad Nayyer Zubair Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Stony Brook University, 2004

[2] Junjiro Okajima Another UnionFS http://aufs.sf.net, 2006

[3] Jan-Simon Pendry, Marshall Kirk McKusick Union Mounts in 4.4BSD-lite In *Proceedings of the USENIX 1995 Technical Conference*, page 3. USENIX Associations, 1995.

[4] David Hendricks, Evan Adams, Tom Lyon, Terrence C. Miller. Method and Apparatus for Translucent File System United States Patent 5,313,646, May 17, 1994

[5] Werner Almsberger Inheriting File System http://icapeople.epfl.ch/almsber/ifs.html, 1993

[6] Dobrica Pavlinusic, Csaba Henk, Miklos Szeredi, Richard Dawe, Sebastien Delafond Filesystem in Userspace http://fuse.sourceforge.net/, 2008

[7] Goldwyn Rodrigues Unifying Filesystems with Union Mounts http://lwn.net/Articles/312641/ December 24, 2008

[8] Valerie Aurora Unioning File Systems: Architecture, Features, and Design Choices http://lwn.net/Articles/324291/ March 18, 2009

[9] The python language. http://www.python.org/.

[10] Various Merge Algorithms http://revctrl.org/CategoryMergeAlgorithm/

[11] Daniel P. Bovet, Marco Cesati The Virtual Filesystem *Understanding the Linux Kernel*, 303-342, September 2000