A File System Component Compiler

Erez Zadok
ezk@cs.columbia.edu

Computer Science Department
Columbia University
New York, NY 10027

December 7, 1999

THESIS PROPOSAL

File System development is a difficult and time consuming task, the results of which are rarely portable across operating systems. Several proposals to improve the vnode interface to allow for more flexible file system design and implementation have been made in recent years, but none is used in practice because they require costly fundamental changes to kernel interfaces, only operating systems vendors can make those changes, are still non-portable, tend to degrade performance, and do not appear to provide immediate return on such an investment.

This proposal advocates a language for describing file systems, called FiST. The associated translator can generate portable C code — kernel resident or not — that implements the described file system. No kernel source code is needed and no existing vnode interface must change. The performance of the file systems automatically generated by FiST can be within a few percent of comparable hand-written file systems. The main benefits to automation are that development and maintenance costs are greatly reduced, and that it becomes practical to prototype, implement, test, debug, and compose a vastly larger set of such file systems with different properties.

The proposed thesis will describe the language and its translator, use it to implement a few file systems on more than one platform, and evaluate the performance of the automatically generated code.

Contents

1 Introduction **1**

2 Background **4**

3 Mechanisms for Interposition and Composition **15**

4 The FiST Language **24**

5 Evaluation Plan **39**

6 Related Work **41**

7 Summary **46**

A Appendix: Vnode Interface Tutorial **47**

B Appendix: Typical Stackable File Systems **57**

C Extended Examples Using FiST **63**

D Appendix: Wrapfs Mount Code **72**

E Appendix: Portability Using Autoconf **77**

List of Figures

1 A Complex Composed File System 1
2 Data Path in a Device Level File System 4
3 Data Path in a User Level File System 5
4 Data Path in a Vnode Level File System 7
5 Typical Propagation of a Vnode Operation in a Chained Architecture 10
6 Composition Using Pvnodes 11
7 Interposition Resulting in Fan-in or Fan-out 15
8 Private Data of an Interposing Vnode 15
9 Data Structures Set for a Caching File System 16
10 Skeleton Create Operation for the "Wrap" File System Type 17
11 Wrapfs Vnode Interposition and Composition Code 18
12 Skeleton Getattr Operation for the "Wrap" File System Type 19
13 Private Data Held for Each Interposing VFS 20
14 FiST Grammar Outline 30
15 FiST Default Rule Action for Stateless and In-Core File Systems (Pseudo-Code) 32
16 FiST Default Rule Action for Persistent File Systems (Pseudo-Code) 33
17 Vnode Structure in a Stateless File System 34
18 Vnode Structure in a Persistent File System 35

19 Fan-Out in Stackable Vnode File Systems . . . 37
20 Fan-Out in Stackable NFS File Systems . . . 38
21 SunOS 5.x VFS Interface . . . 47
22 SunOS 5.x VFS Operations Interface . . . 48
23 VFS Macros . . . 49
24 VFS Macros Usage Example . . . 49
25 SunOS 5.x Vnode Interface . . . 50
26 SunOS 5.x Vnode Operations Interface . . . 51
27 Some Vnode Macros . . . 54
28 Vnode Macros Usage Example . . . 54
29 File System Z as Y mounted on X . . . 55
30 FiST Definition for Crossfs . . . 63
31 Vnode Code Automatically Generated by FiST for Crossfs . . . 64
32 NFS Code Automatically Generated by FiST for Crossfs . . . 64
33 FiST Definition for Gzipfs . . . 65
34 Vnode Code Automatically Generated by FiST for Gzipfs . . . 66
35 NFS Code Automatically Generated by FiST for Gzipfs . . . 66
36 FiST Definition for Replicfs (top) . . . 67
37 FiST Definition for Replicfs (reading operations) . . . 68
38 FiST Definition for Replicfs (writing operations) . . . 69
39 Vnode Code Automatically Generated by FiST for replicfs (reading operation) . . . 70
40 Vnode Code Automatically Generated by FiST for replicfs (writing operation) . . . 71
41 VFS Sample Code Using Autoconf . . . 78

List of Tables

1 The Four-Space of File Systems . . . 8
2 Dominant File Systems and Code Sizes for Each Medium . . . 12
3 NFS V2 Equivalent Vnode Operations . . . 22
4 NFS V3 Equivalent Vnode Operations . . . 23
5 FiST Vnode Primary Attributes . . . 26
6 FiST Kernel Global State Attributes . . . 26
7 FiST Meta Functions . . . 27
8 FiST Declaration Keywords . . . 31
9 Code Section Names for Stateless and In-Core File Systems . . . 32
10 Code Section Names for Persistent File Systems . . . 33

1 Introduction

A "vnode" is a data structure used within Unix-based operating systems to represent an open file, directory, device, or other entity (e.g., socket) that can appear in the file system name-space. The "vnode interface" is an interface within an operating system's file system module. It allows higher level operating system modules to perform operations on vnodes. The vnode interface was invented by Sun Microsystems to facilitate the coexistence of multiple file systems [Kleiman86], specifically the local file system that manages disk storage and the NFS [Sun89, Pawlowski94] remote file system. When a vnode represents storage (such as a file or directory), it does not expose what type of physical file system implements the storage. This "virtual file system" concept has proven very useful, and nearly every version of Unix includes some version of vnodes and a vnode interface.

One notable improvement to the vnode concept is "vnode stacking," [Rosenthal92, Heidemann94, Skinner93] a technique for modularizing file system functions. The idea is to allow one vnode interface to call another. Before stacking existed, there was only a single vnode interface. Higher level operating systems code called the vnode interface which in turn called code for a specific file system. With vnode stacking, several vnode interfaces may exist and they may call each other in sequence: the code for a certain operation at stack level N calls the corresponding operation at level $N + 1$, and so on.

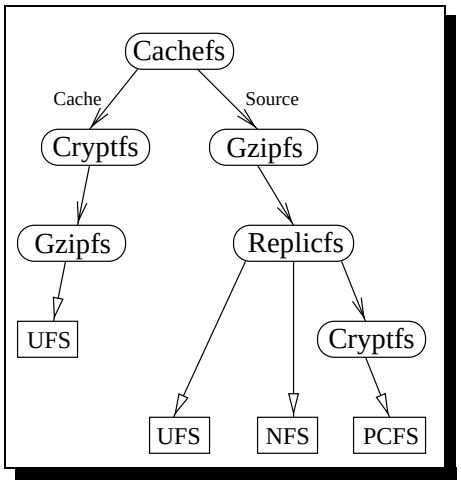

Figure 1: A Complex Composed File System

For an example of the utility of vnode stacking, consider the complex caching file system (Cachefs) shown in Figure 1. Here, files are accessed from a compressed (Gzipfs), replicated (Replicfs), file system and cached in an encrypted (Cryptfs), compressed, file system. One of the replicas of the source file system is itself encrypted, presumably with a key different from that of the encrypted cache. The cache is stored in a UFS [LoVerso91] physical file system. Each of the three replicas is stored in a different type of physical file system, UFS, NFS, and PCFS [Forin94].

One could design a single file system that includes all of this functionality. However, the result would probably be complex and difficult to debug and maintain. Alternatively, one could decompose such a file system into a set of components:

1. A caching file system that copies from a source file system and caches in a target file system.
2. A cryptographic file system that decrypts as it reads and encrypts as it writes.
3. A compressing file system that decompresses as it reads and compresses as it writes.
4. A replicated file system that provides consistency control among copies spread across three file systems.

These components can be combined in many ways provided that they are written to call and be callable by other, unknown, components. Figure 1 shows how the cryptographic file system can stack on top of either a physical file system (PCFS) or a non-physical one (Gzipfs). Vnode stacking facilitates

this design concept by providing a convenient inter-component interface. The introduction of one module on top of another in the stack is called "interposition."

Building file systems by component interposition carries the expected advantages of greater modularity, easier debugging, scalability, etc. The primary disadvantage is performance. Crossing the vnode interface is overhead. However, I claim that the overhead can be made so small that any loss in performance is outweighed by the benefits. See Section 3.6.2.

The example in Figure 1 illustrates another property of vnode stacking: fanout. The implementation of Replicfs calls three different file systems. Fan-in can exist, too. There is no reason to restrict the stacking concept to a linear stack or chain of file systems.

1.1 The Problem

Despite the promise of vnode stacking, not one of several proposed implementations [Rosenthal90, Rosenthal92, Heidemann94, Skinner93] has made it into mainstream operating systems, even though several of the proposals were made by an operating system vendor (Sun Microsystems).

All previous proposals for vnode stacking required substantial changes to the definitions of the vnode and the vnode interface. These proposals did not meet with wide acceptance, for a few reasons:

- The need to modify operating system source code. Only vendors, not individual researchers, could make the necessary changes to commercial operating systems.
- The large cost of installing a significant change to an operating system, and the corresponding concern about return on investment. It was not obvious to vendors that support for vnode stacking would result in better short-term sales.
- The concern that generalization of the vnode interface might harm performance.

Additionally, the vnode stacking proposals have always been linked to a particular operating system, and hence unportable. For more details see Section 2.3.2.

Although these objections to the vnode stacking concept are not technical, they are fundamental. However, this thesis proposal will present another technical solution which avoids most of the above problems.

1.2 My Solution

My thesis is that it is possible to implement vnode stacking in a fashion that is portable across operating systems, without the need for kernel source code, without having to change the existing vnode interface, and with only a negligible decrease in file system performance.

In particular, I propose to demonstrate a language and compiler called *FiST* (for File System Translator). New types of file systems — such as the compressing or encrypting file systems mentioned above — are described as FiST programs. The FiST compiler then produces C code matched to the internal interfaces of the target operating system. Besides ease of implementation and portability, a further advantage of implementing file systems in FiST is that the resulting C code can be tailored for either in-kernel or out-of-kernel use, according to whether performance or flexibility is the primary goal.

1.3 Advantages of My Solution

The advantages of a file system compiler are:

1. It should be easier to implement a file system in FiST than in C.
2. FiST adapts to the existing vnode definitions, so a single FiST program can be compiled to run on different operating systems that may have different vnode definitions.
3. No kernel sources (usually proprietary and expensive) are required, meaning that anyone, not just operating system vendors, can become a file system implementer. Eliminating the need for kernel code also saves time spent on browsing and modifying, and licensing costs for *each* file system and platform.
4. The same FiST program can be used to generate kernel-resident or user-level modules. User-level file systems are useful during development, as they are much easier to debug. Kernel-level file systems provide the best performance.

I expect that FiST will substantially increase the ease and speed with which researchers can prototype new file system ideas, thereby leading to a qualitative improvement in file system innovation.

1.4 Organization of this Proposal

Can file systems perform adequately and be source-portable at the same time? The answer is yes, and is explored in the rest of this proposal. Section 2 provides background on vnode interfaces proposed and used over the past decade. Section 3 explains the conceptual design of my system, including the current implementation. Section 4 details the core of this proposal: the FiST language (with a few extended examples in Appendix C). I map out the plan for evaluating my work in Section 5 and describe related work in Section 6. I conclude with a summary in Section 7.

Several appendices follow, expanding on relevant material. These include a tutorial on vnodes for readers not familiar with the details, a list of example file systems that could be generated using FiST, a set of extended examples using FiST and showing code that would be generated, actual working code showing the actions that occur when a file system is stacked on top of another, and the last appendix describes facilities for promoting portability.

2 Background

There are many operating systems, and many new file systems have been proposed, but only a handful of file systems are in regular use. This section provides a brief history of the evolution of file systems in general and the vnode interface in particular, and attempts to explain why so few file systems are used in practice. To a large degree, the reasons overlap with the limitations that FiST is intended to remove.

2.1 Types of File Systems

I classify file systems into three categories, based on how they are accessed: device level, out of kernel, and vnode level.

2.1.1 Device Level

The lowest level file systems are part of the operating system and call device drivers directly. These file systems are usually aware of and often optimized for specific device characteristics, as shown in Figure 2.

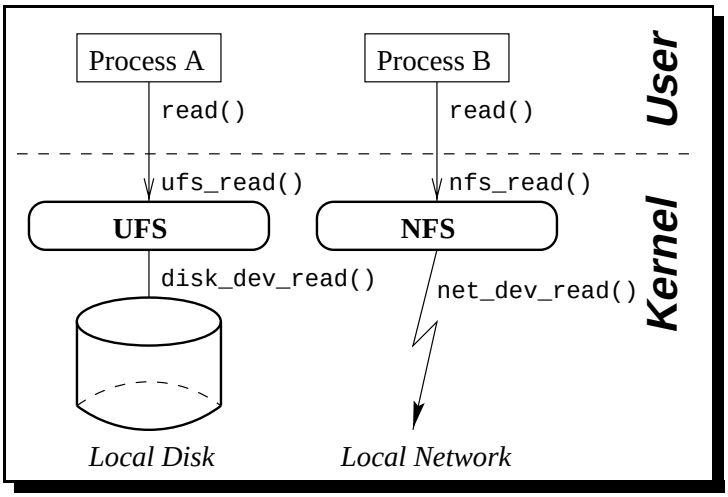

Figure 2: Data Path in a Device Level File System

Examples of such file systems include

- The Berkeley Fast File System (FFS) [McKusick84] for physical disks.
- Sun Microsystem's UFS [LoVerso91], an optimized version of FFS.
- The LFS "log structured" file system, optimized for sequential writes [Rosenblum91] on hard disks.

- NFS [Sandberg85, Pawlowski94], that uses the network as its file system "device."[1]
- The High-Sierra file system (HSFS, ISO9660) for CD-ROMs [Kao89].
- The FAT-based file system originally developed for DOS [Tanenbaum92], and later adapted for Unix machines to access a floppy as a native PC-based file system (PCFS) [Forin94].

Such file systems are difficult to port because they are coupled to the surrounding operating system: system call handlers call the file system code and the file system code calls device drivers.

Because these file systems are optimized for the common combination of hard disks and Unix workloads, we find only a handful in use. Note that while many Unix vendors have their own version of a disk-based local file system, these are in most cases only small variations of the Berkeley FFS.

2.1.2 Out of Kernel

The highest level file systems reside outside the kernel. They are implemented either as a process or as a run-time library. Most such file systems are accessed via the NFS protocol. That is, the process that implements them registers with the kernel as an NFS server, although the files it manages are not necessarily remote.

The primary benefits of user-level file systems are easier development, easier debugging, and portability. However, user level file systems suffer from inherently poor performance. Figure 3 shows how many steps it takes the system to satisfy an access request through a user-level file server. Each crossing of the dashed line requires a context switch and, sometimes, a data copy.

Additionally, user level implementation raises the danger of deadlock, as the process implementing the file system must interact with the operating system, sometimes regarding the very file system it is implementing. Finally, user level implementation creates new failure modes. If there is a bug in a kernel-resident file system, the system will crash; though highly undesirable, this is the familiar "fail-stop"

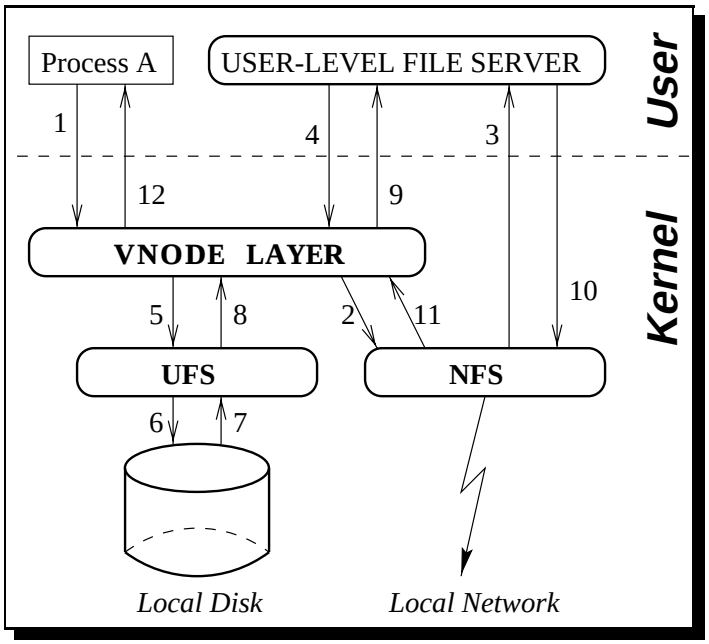

Figure 3: Data Path in a User Level File System

[1] While NFS does not use the network as a persistent storage medium, it uses it to communicate to servers that, in turn, store the files on local disks.

[2] For example our department uses the Amd automounter and has seen more than once the severe effects of its aborting. Shells hang because of attempts to access auto-mounted paths in users' $PATH variables, so no new programs can be started. Long-running programs such as **emacs** also hang because they often perform file systems access for user files, auto-saves, auto-loading emacs-lisp files, etc. Within a few minutes of Amd's abnormal exit, the machine becomes unusable and needs a reboot.

failure model. In contrast, when an out
of kernel file system hangs or exits, processes that access the now-dead file system live on, possibly propagating erroneous results to other processes and machines.[2]

Examples of out-of-kernel file systems are the Amd [Pendry91, Stewart93] and Automountd [Callaghan89] automounters, Blaze's Cfsd encrypting file system [Blaze93], and Amd derivatives including Hlfsd [Zadok93b], AutoCacher [Minnich93], and Restore-o-Mounter [Moran93].

A few file systems at the user level have been implemented as a user-level library. One such example is Systas [Lord96], a file system for Linux that adds an extra measure of flexibility by allowing users to write Scheme code to implement the file system semantics. Another, also for Linux, is Userfs [Fitzhardinge94]. For example, to write a new file system using Userfs, the implementor fills in a set of C++ stub file system calls — the file system's version of `open`, `close`, `lookup`, `read`, `write`, `unlink`, etc. Developers have all the flexibility of user level C++ programs. Then, they compile their code and link it with the provided Userfs run-time library. The library provides the file system driver engine and the necessary linkage to special kernel hooks. The result is a process that implements the file system. When run, the kernel will divert file system calls to the custom-linked user-level program they just linked with.

Such flexibility is very appealing. Unfortunately, the two examples just mentioned are limited to Linux and cannot be easily ported to other operating systems because they require special kernel support. Also, they still require the user to write a full implementation of each file system call.

2.1.3 Vnode Level

As mentioned in the introduction, some file systems are implemented as in-kernel modules that export the vnode interface. They typically implement "meta" operations on files or groups of files, relying on other device level file systems for file access. Examples include Solaris' Cachefs [SunSoft94], and the Online Disk-Suite (OLDS) of file systems (offering mirroring, striping, and device concatenation) [SMCC93b].

For example, the mirroring file system of the Online Disk-Suite is a module that stacks on top of two or more physical file systems. Each vnode operation in the mirroring file system performs "meta" operations on the native file systems it stacked on top of. For example, the `read` call reads data from either one of the replicas and returns the first one that replies; the `write` call writes data to all replicas and will not return a success status until the data have been successfully written to all copies.

To access a particular file system, processes make system calls that get translated into vnode interface calls, as depicted in Figure 4.

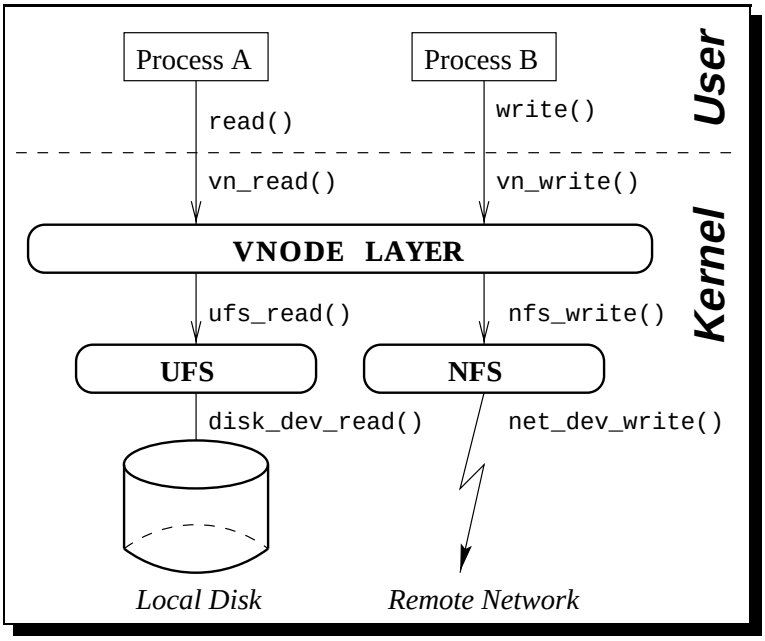

Figure 4: Data Path in a Vnode Level File System

Vnode level file systems exhibit the same advantages and disadvantages as device level file systems, though to a lesser degree. Kernel residence makes writing such file systems difficult, but their performance is good.

The FiST compiler produces code for either a vnode level file system or one running at user level. The first reason for choosing the vnode level over device and user levels is that most proposals for new file systems are proposals for "meta" semantics rather than new ways to organize bits on devices. The second reason is the possibility of good performance because a kernel-resident implementation avoids costly context switches, and runs in a higher privileged mode than user level. The third reason is the potential for portable code because most brands of Unix implement some version of the vnode interface.

Debugging kernel resident file systems is still difficult. For that reason, I decided that FiST will also generate file system modules to run in user level, where they can be inspected with greater ease using standard debuggers. These modules will be generated to an API that is supported by Amd (NFS). Amd will be able to dynamically load these file systems and provide new semantics based on the FiST descriptions thereof.

Supporting both user and kernel level file systems from the *same* FiST description provides the best of both worlds: you get (a) good performance when running in the kernel, and (b) easier development when running modules in Amd.

2.1.4 Compilation vs. Interpretation

Another dimension — beside in-kernel versus out-of-kernel — for categorizing file systems is whether the functionality is compiled or interpreted. Table 1 summarizes the advantages and disadvantages of these four possibilities.

Location	Language	Advantages	Disadvantages	Examples	FiST
Kernel	Compiled	Best performance.	Difficult to write.	NFS, UFS, and most others.	Yes
User-level	Compiled	Easier to write. Better performance than interpreted.	Worse performance than in kernel due to context switching.	Amd and Sun's Automounter.[a]	Yes
Kernel	Interpreted	Does not make much sense, since performance is the main reason to move into the kernel.	Interpreted language would seriously degrade performance. Kernel is still a "hostile" environment.	None, possibly Java based systems.[b]	No
User-level	Interpreted	Very easy to write and modify. Easy to debug in user-level.	Poorest performance due to interpretation and context switches.	Systas	No

[a] These automounters do not contain all of their file system functionality in C code, only the base part. Additional functionality is provided by static configuration maps the administrator has to edit by hand.

[b] Note that Java is not a strictly interpreted language, but a byte-compiled one.

Table 1: The Four-Space of File Systems

FiST will be able to generate compiled code for either user-level or the kernel. This results in both speed (compiled, in-kernel) and ease of development and debugging (user-level). Since FiST is a higher-level language it would allow relatively easy changes to file systems, the same way interpreted languages do.

2.2 The Vnode Interface

2.2.1 The Original Vnode Interface

The vnode interface[3] was invented over a decade ago to facilitate the implementation of multiple file systems in one operating system [Kleiman86], and it has been very successful at that. It is now universally present in Unix operating systems. Readers not familiar with the vnode interface may refer to Appendix A for a tutorial on the subject.

The designers of the original vnode interface envisioned "pluggable" file system modules [Rodriguez86], but this capability was not present at the beginning. Through the 1980s Sun made at least three revisions of the interface designed to enhance plugability [Rosenthal90]. However, during the same period

[3] When I speak of the "vnode interface," it should be taken to include the vnode interface that provides operations on files *and* the VFS interface that provides operations on file systems.

Sun lost control of the vnode definition as other operating system vendors made slight, incompatible, changes to their vnode interfaces.

2.2.2 A Stackable Vnode Interface

We recognize that one-size-fits-all file systems are insufficient in many cases. Specialized file systems are often proposed but rarely implemented. Four example domains include:

1. Multimedia: with the explosion of the Internet, Web content developers would like a file system that can store HTML, image, and audio files more efficiently so they can be retrieved faster with HTTP servers, or be played back in real-time [Anderson92, Ramakrishnan93, Fall94, Mercer94, Pasquale94].

2. Databases: researchers are looking for methods to improve the performance of Unix file systems, and/or for file systems that provide built-in support for concurrency [Stonebraker81, Stonebraker86].

3. Mobility: replicated and distributed file systems with disconnected and caching operations figure heavily in an environment where network latency and reliability is highly variable [Satyanarayanan90, Kistler91, Tait91, Tait92, Kistler93, Zadok93a, Kuenning94, Marsh94, Mummert95].

4. Security: more secure file systems are sought, especially ones that securely export files over the network [Steiner88, Haynes92, Glover93, Takahashi95]. An easy way to use encryption in file systems [Blaze93, Gutmann96, Boneh96] and the ability to provide special semantics via facilities such as general purpose Access Control Lists (ACLs) [Kramer88, Pawlowski94] are also highly desirable [Bishop88, Kardel90].

Researchers and developers have always needed an environment where they can quickly prototype and test new file system ideas. Several earlier works attempted to provide the necessary flexibility. Apollo's I/O system was extendible through user-level libraries that changed the behavior of the application linking with them [Rees86]; now, modern support for shared libraries [Gingell87a] permits new functionality to be loaded by the run-time linker. One of the first attempts to extend file system functionality was "watchdogs" [Bershad88], a mechanism for trapping file system operations and running user-written code as part of the operation.

Vnode stacking was first implemented by Rosenthal (in SunOS 4.1) around 1990 [Rosenthal90]. His work was both the first implementation of the plugability concept and also a clean-up effort in response to changes that had been required to support integration of SunOS and System V and to merge the file system's buffer cache with the virtual memory system. Because it focused on the universally available vnode interface, Rosenthal's stacking model was not ad hoc, unlike earlier efforts, and held promise as a "standard" file system extension mechanism.

With vnode stacking, a vnode now represents a file open *in a particular file system*. If N file systems are stacked, a single file is represented by N vnodes, one for each file system. The vnodes are chained together. A vnode interface operation proceeds from the head of the chain to the tail, operating on each vnode, and aborting if an error occurs. This mechanism, which is similar to the way Stream I/O modules [Ritchie84] operate, is depicted in Figure 5.

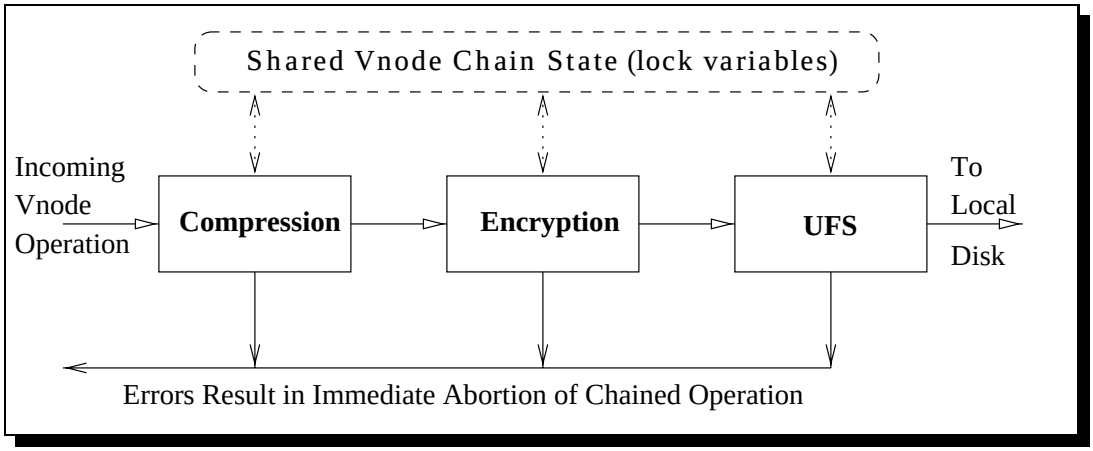

Figure 5: Typical Propagation of a Vnode Operation in a Chained Architecture

This simple interface alone was capable of combining several instances of existing UFS or NFS file systems to provide replication, caching, and fall-back file systems, among other services. Rosenthal built a prototype of his proposed interface in the SunOS 4.1 kernel, but was not satisfied with his design and implementation for several reasons: locking techniques were inadequate, the VFS interface had not been redesigned to fit the new model, multi-threading issues where not considered, and he wanted to implement more file system modules so as to get more experience with the interface. Rosenthal's interface was never made public or incorporated into Sun's operating systems.

A few similar works followed Rosenthal, such as further prototypes for extended file systems in SunOS [Skinner93], and the Ficus layered file system [Guy90, Heidemann91] at UCLA.

2.2.2.1 Interposition and Composition

Later works [Rosenthal92, Skinner93] established the current terminology for the field, discarding "stacking" in favor of "interposition" and "composition." The term "stacking" was considered at once to have too many implications, to be too vague, and to imply only a linear LIFO structure with no fan-in or fan-out.

Interposition is the new term for stacking. The defining papers [Rosenthal92, Skinner93] explain a particular implementation of interposition based on a new definition of vnode. The new vnode contains only the public fields of the old vnode and a new data structure called a *pvnode* contains the private fields of the old vnode. A "vnode chain" now becomes a single vnode (providing a unique identity for the file) plus a "chain"[4] of linked pvnodes. Interposed functionality is represented by one pvnode per open file.

Pvnodes may contain pointers to other vnodes, with the effect that all the linked vnodes may need to be regarded as a single object. This effect is called composition. Composition, in particular, requires the following two capabilities [Rosenthal92]:

1. The ability to lock a complete interposition chain with one operation.

[4] Actually a DAG, to provide fan-in and fan-out.

2. Treating an interposition chain as an atomic unit. An operation that failed midway should result in undoing anything that was done when the operation began at the head of the chain.

Figure 6 shows this structure for a compressing, encrypting file system, that uses UFS as its persistent storage. For each of the three file system layers in the stack, there is one pvnode. Each pvnode contains a pointer back to the file system that it represents, so that the correct operations vector is used. The three pvnodes are linked together in the order of the stack from the top to the bottom. The head of the stack is referenced from a single vnode structure. The purpose of this restructuring that Skinner & Wong had proposed was so that the three pvnodes could be used as one composed entity (shown here as a dashed enclosing box) that could be locked using a single lock variable in the new vnode structure.

The linked data structures created by interposition and the corresponding complex semantics arising from composition complicate concurrency control and failure recovery.

One concurrency control problem is how to lock an arbitrarily long interposition chain as cheaply as possible. Another, harder, problem is how to lock more than one chain for multi-vnode operations.

The failure recovery problem arises from composition. If a multi-vnode operation fails midway, it is vital to rollback the operations that have succeeded. Both Rosenthal and Skinner & Wong discuss adapting the database concept of atomic transactions. Specifically, each pvnode would contain routines to abort, commit, and "prepare"[5] the effects of operations on it. However, probably because of the complexity involved, no one has yet implemented transactions in support of composition. Consequently, "stacks" of interposed file systems may have failure behavior that is different from single file systems.

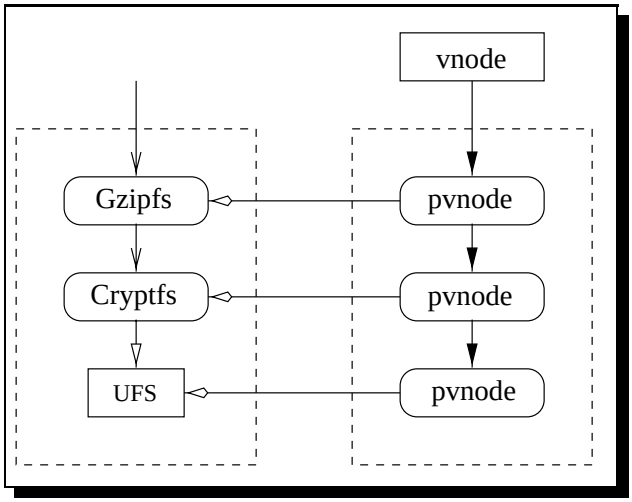

Figure 6: Composition Using Pvnodes

2.3 Barriers to File System Experimentation

2.3.1 Inertia

An interesting observation, seen in Table 2, is that each device level file system listed in Section 2.1.1 has only a handful, generally no more than two, dominant implementations in use for each storage medium.

One might wonder why this is the case. Is it pure luck that these were the first file systems ever implemented on these media, and now they are the de-facto standards for their media? I think there are several reasons for their dominance. They are dominant because they are *good* and they satisfy the needs of the majority of users. They have been adapted to observed workload and improved quite a bit over the years, to a point where anyone thinking of writing a new one at that level has to come

[5] In transaction terminology, "prepare" means to stop processing and prepare to either commit or abort.

Media Type	Dominant File System	Avg. Code Size (C lines)	Other File Systems
Hard Disks	UFS (FFS)	20,000	LFS
Network	NFS	30,000	Amd, AFS
CD-ROM	HSFS (ISO-9660)	6,000	UFS, CD-I, CD-V
Floppy	PCFS (DOS)	6,000	UFS

Table 2: Dominant File Systems and Code Sizes for Each Medium

up with something substantially better to get a sizable share of the market. In short, reasons to write new file systems at the device level are rarely compelling, or surely they would have been written.

Every file system (or kernel) developer would agree that writing a new file system takes a long time, is difficult to test and debug, and has to be constantly maintained as operating systems evolve and change. Every small change takes a long edit-compile-run-debug cycle, with kernel crashes and lack of debugging tools [Golub90, Stallman94, SMCC94a] making the task frustrating. Worse, file system code developed for one operating system is almost never portable to another. After a long period of development for one operating system, the whole process has to be repeated if the file system is to be ported to a new operating system. It should come as no surprise, given the sheer size of file system code, that vendors and independent software vendors (ISVs) are reluctant to develop new file systems, at least at the device level.

2.3.2 Commercial Concerns

Given the history of the vnode interface I find it curious why the chief advocate of vnode interposition (judging by the number of papers on the subject), Sun Microsystems, has not included any fundamentally new vnode interface in their operating systems. Sun has released over half a dozen new versions of their Solaris operating system in the past few years, so they certainly had the opportunity to include a new interface had they wanted to.

I've had several personal communications with experts in the field: Brent Callaghan, Glenn Skinner[6] and a few others who chose to remain anonymous. Unanimously, they told me that while they thought that vnode interposition is desirable, more pressing projects were given higher priority. They cited management concerns over the commerciability of stackable file systems, the overall cost of making such radical changes to the operating system, and the perceived lack of short-term benefit from making such changes. In addition, management did not want to incorporate any changes that degraded performance even slightly in the then-fledgling Solaris 2.x operating system.

2.3.3 High Development Costs

I have had seven years of personal experience in writing, porting, maintaining, and modifying file systems — including NFS, Amd, Hlfsd, and LFS — and ranging across several operating systems. From this experience, I know that while there is a "nearly standard" vnode interface, writing or porting a file system to that interface is a substantial task. In addition, file system code is usually tightly coupled internally; many operations depend on others within the same file system. The work cannot be easily

[6] Both currently working at Sun Microsystems.

parallelized to a large group, and one often finds that a single file system is written by one person. For example, most of the file systems written for Linux have been initially written by individuals. It is often the same small set of developers that develop all the different file systems for an operating system. This is further corroborated from inspection of sources for many public and commercial file systems; I have frequently noted the coding style to be different from one file system to another, while RCS tags and "Change Log" entries within the same file system repeatedly made by the same person.

I concluded that, of all possible reasons for the limited diversity of file systems, the most compelling one is the complexity and time involved in overhauling an operating system and all its file systems to a new, albeit better, vnode interface.

Therefore I decided to try to provide vnode interposition and composition capabilities, in a portable way, *without* requiring kernel sources, and more importantly, *without* changing existing vnode interfaces. The next section explains my approach, FiST, as the next logical step in the evolution of extensible file systems.

2.4 FiST

The effort to change the vnode interface was driven by the need to simplify the model, and allow new file systems to be written faster. This was partially done by removing old vnode calls such as `vn_bread` and adding new ones such as `vn_map` [Rosenthal90]. Changing the vnode interface was akin to changing the "language" with which a file system implementor "spoke" with the kernel. Several past works — such as Skinner and Wong's "Interposer Toolkit" — began to address the issue of describing file systems using a higher-level language. The most successful of all is the simple (albeit limited) language used by Amd [Pendry91, Stewart93] to describe map entries, their types, semantics, etc. Recent work on file system simulators [Bosch96] also moves in this direction, but unfortunately requires a radically different (object oriented) file system interface.

It was natural then to try to find a better language that can describe file systems at a high level, for the following reasons:

- There is a lot of repetition in file system code. Much of the code for file systems in the same operating systems share the same structure, calling conventions, error handling, and more. A translator could reuse code, or generate similar code from templates.

- There are many tedious details that must be maintained, which file system implementors may forget or neglect. For example there are many calls in the vnode interface that are rarely used, yet need to be implemented. A language translator is perfect for offering default actions for any vnode operation that need not be implemented, taking care of basic error handling, and other mundane tasks.

- Generated code will be bug-free. This can reduce debugging and maintenance time greatly. A bug in a kernel resident module often results in system panics, corrupt file systems, and a lengthy reboot.

- If sufficiently abstract, a single file system description can be used to generate either kernel-resident or user-level code. This lets developers maintain and debug a new file system at user level, then move the code into kernel level only when they feel that it is stable enough. Meanwhile

applications and utilities can be designed, developed, and tested using the user-level version of the new file system.

- An interposeable file system module typically cannot be binary or source portable because of the different facilities offered by different operating systems. A higher level description can offer portability.
- Maintaining file systems through a higher level language becomes easier. Changes to features can be localized into a few places in a description, whereas when writing file system code directly, the same changes have to be updated and verified in numerous places.

3 Mechanisms for Interposition and Composition

I have conducted a feasibility study by implementing, by hand, in Solaris 2.4, much of the code that FiST will have to produce automatically. This section explains the operation of that code, which will be used as templates by the FiST compiler. Section 4 describes the FiST language.

3.1 Interposition API

The `mount` system call is used to interpose one file system on top of another. Likewise, `umount` unmounts and hence de-interposes a file system. Mounting can be relative to any file system above or below, so that file systems can be "stacked" into a DAG. As an example, suppose that file system X is interposed on file system Y. To create fan-in, file system Z can be mounted above Y. To create fan-out, Z can be mounted below X.

Figure 7 shows what mounts result in a fan-in vs. a fan-out. The information of how many file systems are mounted at a mount point is stored in a private VFS data structure, and is described in Section 3.5.

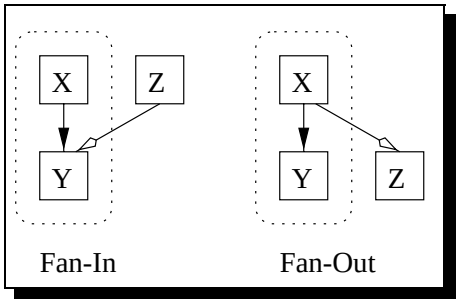

Figure 7: Interposition Resulting in Fan-in or Fan-out

3.2 Creating Links Among Vnodes

For each open file or directory, a vnode is allocated for each level of interposition. If there have been N interpositions, then an open file will have $N + 1$ associated vnodes, one for the "base" file system and one for each file system that has been interposed.

Each vnode of an interposing file system must have access to the vnode(s) that it interposes upon, and this must be accomplished without changing the vnode definition. Fortunately, the vnode contains a pointer (called `v_data`) to an opaque private area. For each type of file system (e.g., NFS, UFS) this pointer may point to a structure that includes extra information needed by that type of file system.

```
typedef struct fist_wrapnode {
  vnode_t *  fwn_vnodep[];   /* pointers to interposed-on vnodes */
  int        count;          /* # of pointers; >1 indicates fanout */
  /* additional per-vnode data here */
} fist_wrapnode_t;
```

Figure 8: Private Data of an Interposing Vnode

I have created a new "wrap" vnode type for interposing vnodes. For this type of vnode, `v_data` points to the private `fist_wrapnode_t` data structure shown in Figure 8. This structure contains pointers to the vnodes it interposes upon. Pointers link vnodes from the top down, representing the DAG of interposed file systems for each open file. Figure 9 shows an example of a caching file system, and the relationship between the vnodes in use and their respective private data, especially that of the interposing file system.

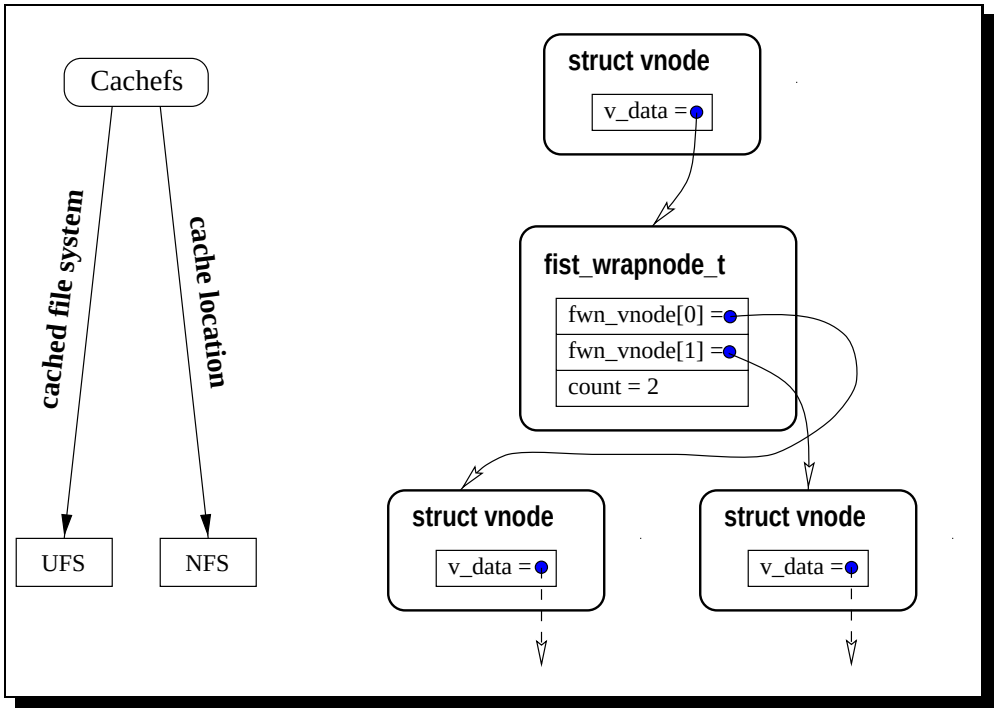

Figure 9: Data Structures Set for a Caching File System

These pointers are established when a vnode is created. There are only five vnode-creating operations, one of which is "`create`." I have altered the code for each operation as suggested by Figures 10 and 11. (The other functions that create new vnodes, and therefore use the interposition code in Figure 11 are `lookup`, `mkdir`, `open`, and the utility routine `realvp`. The VFS function `vfs_mount` also creates a new vnode, the root vnode of the file system.)

Figure 10 is a skeleton of the actual code for the vnode operation `create` in Wrapfs.[7] It shows what happens when creating an entry called `name` in the directory represented by the vnode `*dvp`, assuming that there is no fanout. The first line obtains a pointer to the vnode that `*dvp` interposes upon. The second line calls the `VOP_CREATE` macro. (Solaris and most VFS implementations have `VOP_*` macros whose purpose is to hide the type of underlying file system that is performing the operation.) The macro expands to invoke the create operation in the operations vector for whatever type of vnode `*hidden_vp` is. If it is another interposing vnode of the Wrapfs file system, then `fist_wrap_create` will be called recursively; otherwise, the create operation for the appropriate type of file system will be called. The third line calls `fist_wrap_interpose()`, which sets up three important pointer fields. This logic can be extended simply when there is fanout.

Vnode destruction is handled similarly. Vnodes are deallocated when their reference count reaches zero; then, the `inactive` operation is called. The interposing file system simply calls the inactive operation on the interposed file system and then deallocates the current vnode.

[7] I do not change system call interface functions such as `create(2)`; the kernel is responsible for calling a particular file systems' vnode operations, and those I may modify.

```
static int fist_wrap_create(vnode_t *dvp, char *name, vattr_t *vap,
                            vcexcl_t excl, int mode, vnode_t **vpp,
                            cred_t *cr)
{
  int error = 0;
  vnode_t *hidden_vp;

  /* get interposed-on vnode */
  hidden_vp = vntofwn(dvp)->fwn_vnodep[0];

  /* pass operation down to interposed-on file system */
  error = VOP_CREATE(hidden_vp, name, vap, excl, mode, vpp, cr);

  /* if no error, interpose vnode */
  if (!error)
    *vpp = fist_wrap_interpose(*vpp, (*vpp)->v_vfsp);

  return(error);
}
```

Figure 10: Skeleton Create Operation for the "Wrap" File System Type

The three pointer fields set up by `fist_wrap_interpose()` are:

1. The interposing vnode's pointer to its private area.

2. The private area's pointer to the vnode that is being interposed upon.

3. The interposing vnode's operation vector.

Just as there are specialized operation vectors for vnodes of types NFS and UFS (`nfs_vnodeops` and `ufs_vnodeops`, respectively), there is also a specialized vector for interposing vnodes. As shown in Figure 11, an interposing vnode has its operations vector field (`v_op`) set to `&fist_wrap_vnodeops`, a set of pointers to functions that provide "wrapping" implementations of vnode interface functions. These functions define a file system that I call Wrapfs.

3.3 Using Links Among Vnodes

Most other vnode/VFS operations in Wrapfs are very simple: they call the operation on the interposed vnode and return the status code. Figure 12 sketches the wrapping implementation of the "Get File Attributes" operation, `getattr`. The pattern is similar to that shown in Figure 10 but differs in that `fist_wrap_interpose()` is not called. In the case of `getattr`, all that happens is that `VOP_GETATTR` expands into the `getattr` operation appropriate for the type of vnode that is interposed upon. Before and after invoking the vnode operation on the interposed vnode, it is possible to manipulate the arguments passed or results returned.

```
/* interpose on an old vnode and return new one */
static vnode_t *fist_wrap_interpose(vnode_t *old_vp, vfs_t *old_vfsp)
{
  vnode_t *new_vp;
  fist_wrapnode_t *wp;          /* private area for vnode_vp */
  fist_wrapinfo_t *ip;          /* private area for vfs */

  /* allocate new vnode */
  new_vp = kmem_alloc(sizeof(vnode_t), KM_SLEEP);
  if (!new_vp)
    return(NULL);
  /* VN_INIT2 is like VN_INIT but reuses v_lock field of interposed vnode */
  VN_INIT2(vp, old_vfsp, old_vp->v_type, (dev_t) NULL, old_vp->v_lock);

  /* allocate vnode's private area */
  wp = (fist_wrapnode_t *) kmem_alloc(sizeof(fist_wrapnode_t), KM_SLEEP);
  if (!wp)
    return(NULL);

  /* set pointers and operations vector */
  new_vp->v_data = (caddr_t) wp;
  wp->fwn_vnodep[0] = old_vp;
  new_vp->v_op = &fist_wrap_vnodeops;

  /* see "Private VFS State" Section for explanation */
  ip = vfstofwi(old_vfsp);
  ip->fwi_num_vnodes++;

  /* return new vnode */
  return(new_vp);
}
```

Figure 11: Wrapfs Vnode Interposition and Composition Code

```
static int fist_wrap_getattr(vnode_t *vp, vattr_t *vap,
                             int flags, cred_t *cr)
{
  int error = 0;
  vnode_t *hidden_vp;

  /* get interposed-on vnode */
  hidden_vp = vntofwn(vp)->fwn_vnodep[0];

  /* Note: can manipulate passed arguments here */

  /* pass operation to interposed-on file system and return status */
  error = VOP_GETATTR(hidden_vp, vap, flags, cr);

  /* Note: can manipulate returning results here */

  return (error);
}
```

Figure 12: Skeleton Getattr Operation for the "Wrap" File System Type

3.4 Composition

As mentioned in Section 2.2.2.1, composition is the term for concurrency control and failure atomicity for an operation that is performed on all the vnodes in an interposition DAG.

My work provides only concurrency control. Pre-existing file system code sets a lock (on a single vnode) at the start of each operation and drops it at the end of the operation. The composition problem is how to extend the control of this lock over all the vnodes in the interposition DAG, without making any changes to existing data structures. Fortunately, the `v_lock` field within a vnode is a pointer to a reference-counted "lock variable" structure. Each time a new vnode is interposed upon an existing one, the interposer's lock field is made another pointer to the lock variable of the interposed vnode. (This code is in macro `VN_INIT2`, which is referenced but not shown in Figure 11.) This technique ensures that all vnodes in a DAG are working with the same lock variable. When the lock is set or dropped, every vnode in the DAG is affected simultaneously. See Sections 3.6.2 and 5.2 for evaluation of the impact of locking on performance.

3.5 Private VFS State

Added state must be attached to each vfs structure (the structure that describes whole file systems) just as for vnodes. The vfs structure also contains a pointer to an opaque private area, so I use the same technique as for vnodes.

An auxiliary `fist_wrapinfo_t` structure, shown in Figure 13, houses a pointer to the vfs structure of the interposed-upon file system and a pointer to the root vnode of the interposing file system. Also, while not strictly necessary, for debugging purposes I added a counter that tracks the number of vnodes in use in the file system.

```
typedef struct fist_wrapinfo {
  struct vfs      *fwi_mountvfs;  /* vfs interposed upon */
  struct vnode    fwi_rootvnode;  /* root vnode */
  int             fwi_num_vnodes; /* # of interposed vnodes */
} fist_wrapinfo_t;
```

Figure 13: Private Data Held for Each Interposing VFS

This background makes it possible to understand the actions taken when an interposing file system is mounted on an interposed-upon file system:

1. Initialize basic fields and assert arguments' validity. One of the important assertions verified is that there are no open files on the mount point and file system being mounted. If there were any, an interposing mount could not ensure that it interposed upon every vnode in the interposed file system.
2. Prepare the private information stored by the interposing VFS.
3. Prepare the private information stored by the interposing vnode. This vnode would become the root vnode for Wrapfs.
4. Fill in the information for the VFS structure. Especially important are the private data held by it (`vfs_data`), the operations vector (`vfs_op`), and the vnode it covers (`vfs_vnodecovered`). See Figure 21 for details of all VFS fields.
5. Allocate a vnode to be used as the root vnode for Wrapfs. Fill in important fields such as the vnode operations vector (`v_op`), the private data field (`v_data`) which stores the interposed vnode, and turn on the `VROOT` flag for that vnode in the `v_flag` field, indicating that this vnode is a root of its file system. See Figure 25 for details of all vnode fields.
6. This root vnode just created is then stored in the private data field of the vfs we are mounting. The VFS operation `vfs_root` is called automatically on a vfs in order to retrieve its root vnode. Storing it in the private data field makes it trivial to return.
7. Indicate in the vnode that is the mount point, that we are mounting this vfs on. This fills in the `v_vfsmountedhere` field of the mount point vnode.
8. Return success or error code.

Appendix D includes the code used to interpose a wrapping module on top of another file system.

3.6 Status of Current Implementation

3.6.1 Portability

One of the reasons for working at the vnode level is to achieve portability that, hopefully, would approach that of user level file systems such as Amd. As of this writing, the Wrapfs code is source-portable

across Solaris 2.3, 2.4, and 2.5 on both the SPARC and x86 architectures. It is also binary compatible across Solaris 2.4 and 2.5 (SPARC architecture). Loadable kernel modules are rarely binary compatible across operating system revisions, as was mentioned in Skinner and Wong's work [Skinner93].

I started this work with proprietary Solaris kernel sources. I extracted from these sources the minimum requirements for building file system modules, and then rewrote the code. At this point, I no longer require any access to proprietary sources.

In addition, I was able to move away from using proprietary build tools. Rather than using Sun's commercial "SPARCcompiler" suite of build tools, I now exclusively use freely available GNU tools such as `gcc`, `gmake`, and the GNU linker and assembler. I was surprised and pleased to find that the latest GNU tools were able to properly build and link Solaris *kernel* modules.

For more details on how I will achieve portability using GNU Autoconf [MacKenzie95], see Appendix E.

3.6.2 Performance

The tests I ran included 24 hours of continuous application of common user programs: `ls`, `du`, `find`, `mkdir` and `rm`. These programs were invoked from a simple driver shell script that ran each one of them in turn. First I ran the script on an unmounted `/usr/local` file system. Then I mounted Wrapfs (once) on top of `/usr/local`, and reran the script. I used the `time` utility to measure how much system time was consumed by each run.

Preliminary performance measurements showed that interposing the Wrapfs file system once on top of UFS resulted in degradation ranging from 3.5% (using Solaris 2.4 x86 on a P90 with 24MB RAM and an IDE disk) to 6.4% (using Solaris 2.4 SPARC on an SS2 with 64MB RAM and a SCSI disk) in reported "system" time.

Therefore, the overhead of the first version of Wrapfs is comparable to the mechanisms implemented by Skinner and Wong [Skinner93] (up to 10%) and the UCLA stackable layers project [Heidemann94] (3%).

3.7 User Level Operation

The FiST compiler can easily generate either kernel-resident or user-level code from the same input. Kernel code implements the vnode interface. User level code implements the NFS interface.

The vnode interface was designed to accommodate version 2 of the NFS protocol. Therefore, there is a straightforward mapping of vnode operations to NFS operations, as shown in Table 3. Accordingly, the same "engine" can easily generate both kernel vnode-layer code and NFS code. See the examples in Appendix C.

Automatically generating code for the latest NFS protocol (version 3) [Pawlowski94] is only marginally more difficult, as can be seen in Table 4. There are several new calls that exist only in version 3 of NFS, but they can be safely ignored *because* there is no direct mapping from a vnode operation to them.

It would be useful to handle NFS V3 as well, and that would mean:

- Modifying Amd to understand the V3 protocol as well as V2. (This effort is already under way.)

No.	NFS V2 Call Name	Vnode/VFS Function	No.	NFS V2 Call Name	Vnode/VFS Function
0	NULL	null (trivial)	9	CREATE	vn_create
1	GETATTR	vn_getattr	10	REMOVE	vn_remove
2	SETATTR	vn_setattr	11	RENAME	vn_rename
3	ROOT	vfs_root	12	LINK	vn_link
4	LOOKUP	vn_lookup	13	SYMLINK	vn_symlink
5	READLINK	vn_readlink	14	MKDIR	vn_mkdir
6	READ	vn_read	15	RMDIR	vn_rmdir
7	WRITECACHE	N/A (rarely used)	16	READDIR	vn_readdir
8	WRITE	vn_write	17	STATFS	vfs_statvfs

Table 3: NFS V2 Equivalent Vnode Operations

- Modifying the FiST language to generate empty stubs for those NFS V3 calls that are being ignored. While not strictly used, they must be implemented, even as calls that will return an error code such as "invalid operation."

Therefore, I plan support NFS V3.

3.7.1 Amd as a User-Level File System Driver

User level code will be linked with Amd, which can serve as a driver for the NFS module in the same way that the kernel serves as one for a stackable vnode module. I will augment Amd with the GNU `libdl` package, a library of calls for using user-level dynamic linking. FiST-produced modules will be automatically and dynamically loaded and unloaded.

There are two major benefits to using Amd.

- Most importantly, I can use normal tools like GDB to debug FiST generated languages as I develop the system. Fixing out-of-kernel bugs is much easier than fixing in-kernel bugs.
- Second, many people know and like Amd, and might be more willing to accept FiST because it is tied to Amd.

As of this writing, much work on Amd was done to prepare it for FiST. I have converted Amd to using GNU Autoconf, and in the process learned much and wrote many useful M4 tests [MacKenzie95]. Amd is near ready to handle FiST generated file system modules.

No.	NFS V3 Call Name	Vnode/VFS Function
0	NULL	null (trivial)
1	GETATTR	vn_getattr
2	SETATTR	vn_setattr
3	LOOKUP	vn_lookup
4	ACCESS	vn_access
5	READLINK	vn_readlink
6	READ	vn_read
7	WRITE	vn_write
8	CREATE	vn_create
9	MKDIR	vn_mkdir
10	SYMLINK	vn_symlink
11	MKNOD	a special version of vn_create
12	REMOVE	vn_remove
13	RMDIR	vn_rmdir
14	RENAME	vn_rename
15	LINK	vn_link
16	READDIR	vn_readdir
17	READDIRPLUS	slightly different version of vn_readdir
18	FSSTAT	vfs_statvfs
19	FSINFO	special version of vfs_statvfs+vn_pathconf
20	PATHCONF	vn_pathconf
21	COMMIT	must be completely written

Table 4: NFS V3 Equivalent Vnode Operations

4 The FiST Language

In this section I detail the motivation, concepts, design, and syntax of the FiST language.

4.1 Motivations for the FiST Language

The motivations for creating the FiST language are as follows:

- Much file system code is repetitive. A language is ideally suited to condense such code into short declarations.
- A language may define defaults for many actions, further reducing the need to hand-write code.
- A translator can ensure that generated code is compilable and bug-free.
- Error testing and reporting can be automated.
- Interfacing to the interposed or interposing file system can be automated.
- Interfacing user level and kernel level code can be automated.

The C preprocessor (`cpp`), in comparison, is not able to conditionally generate sophisticated code. It is more suitable for code expansion from small, static templates.

4.2 Language Requirements

I set forth the following requirements for the FiST language:

- The language should be portable across different operating systems offering the vnode interface, and accommodate small differences in vnode interface implementations.
- The language should have a familiar "look and feel." A model like that used by *yacc* is desirable.
- No tedious or repetitive tasks should be required. Every option that can be automated or defaulted should be.
- There should be keywords that can alter the overall behavior of the generated code. Hopefully, this would make it easy to write a FiST program by altering a working FiST program for a different type of file system.
- On the other hand, the advanced "hacker" should not be left out. There should be facilities to modify or augment the behavior of every vnode operation, from simple keywords all the way to hand-writing C code.
- The language should be as high a level as possible while retaining flexibility to adjust small details and ease of parsing.
- An empty input file should result in a usable file system, in particular the wrapper file system, described in Appendix B.2.1.

4.3 Translator Requirements

I set forth the following requirements for the translator:

- The goal of portability effectively requires that the translator output ANSI C code. In particular, the output should compile with strong error checking such as produced by `gcc -ansi -Wall -Werror`.
- The generated code should not require modifications to existing interfaces and kernel facilities, nor should it attempt to modify existing interfaces or file systems at run time.
- The translator should generate runnable kernel-resident code as described in Section 4.3.1.
- The translator should also be able to generate runnable user-level file system code as described in Section 4.10.
- The translator should generate kernel modules that can be dynamically loaded into a running kernel using facilities such as `modload` [SMCC93a], or linked with other kernel objects to produce a static image of a new kernel [SMCC91]. The latter can then be copied over to the root directory and run when the machine is next rebooted.
- The translator should take the worst-case approach. Any minor problem with the input file or the code generation phase should result in fatal errors. No kernel module should be produced if there is any known chance that it will not run properly.
- Every effort should be made to generate fast code.
- The translator itself should be written using tools and languages that make it easily portable to other environments.

4.3.1 Linkage Requirements for Kernel Modules

Kernel modules do not get fully linked when built because some of the symbols they refer to do not exist anywhere but in a running kernel. Despite this complication, the FiST translator should check for any possible unresolved symbols and warn the user.

The naive way to find out if a kernel module is referring to nonexistent symbols is to load it and link it with the running kernel. If any problems arise, the system may hang or panic and crash.

The standard way to avoid this problem is to link the module at user level with a library that includes a `main()` procedure and dummy definitions for all the symbols that a kernel might export.

To write such a library it is necessary to know all the symbols a kernel exports. Older operating systems (such as SunOS 4.x) allow for kernel memory access through a device called `/dev/kmem`. Through this device a privileged process can "browse" the memory of a running kernel to find symbols. The build procedure for newer operating systems (such as FreeBSD 2.1.x) produces a "kernel library" (e.g., `libkern.a`) and header files that include all the symbols one needs.

Following this method, FiST will include an auto-configuration procedure that must be run only once for each operating system version. This procedure will search for all needed kernel symbols and create code to link with file system modules.

4.4 FiST Vnode Attributes

Each vnode has a set of attributes that apply to it. FiST refers to vnode attributes by prefixing their standard names with a % character. Table 5 lists these common attributes.

Attribute	Meaning
%type	regular files, directories, block devices, character devices, symbolic links, Unix pipes, etc. Operations in FiST could apply to one or more of these vnode types (defined in system headers).
%mode	a file has several mode bits that determine if that file can be read, written, or executed by the owner, members of the group, or all others. Also includes "set" bits (setuid, setgid, etc).
%owner	The user ID who owns the file.
%group	The group ID that owns the file.
%size	The size of the file in bytes or blocks.
%time	"Creation," modification, and last access times of the file — referred to as %ctime, %mtime, and %atime, respectively. Defaults to modification time.
%data	The actual data blocks of the file.
%name	The (path) name of the file. This is the first name that a vnode was opened with (in case a file has multiple names). Since usually Unix does not keep file names stored in the kernel, FiST will arrange for them to be stored in the private data of a vnode if this attribute is used.
%fid	The "File ID" of the file (as computed by `vn_fid`).
%misc	Miscellaneous information about a file that would rarely need to be modified.

Table 5: FiST Vnode Primary Attributes

FiST also includes attributes for certain universal Unix kernel concepts that might be useful in specifying file system operations. These are shown in Table 6.

Attribute	Meaning
%cur_uid	The user ID of the currently accessing process.
%cur_gid	The group ID of the currently accessing process.
%cur_pid	The process ID currently running.
%cur_time	The current time in seconds since the Unix epoch.
%from_host	The IP address of the host from where access to this vnode has been initiated. Use `127.0.0.1` for the local host, and `0.0.0.0` if the address could not be found.

Table 6: FiST Kernel Global State Attributes

4.5 FiST Vnode Functions

Each vnode or VFS has a set of operations that can be applied to it. The most obvious are `%vn_`*op* and `%vfs_`*op*. Here, *op* refers to the respective Vnode and VFS operations as described in Appendices A.4

and A.2. For example, `%vn_getattr` refers to the vnode operation "get attributes," and `%vfs_statvfs` refers to the VFS operation "get file system statistics."

It is often useful to refer to a group of vnode operations as a whole. Generally, a user who wants to perform an operation on one type of data will want that operation to be applied everywhere the same type of data object is used. For example, in Envfs (Appendix B.2.2) environment variables in pathnames should be expanded everywhere pathnames are used, not just, say, in the `vn_open` function. FiST provides meta-function operators that start with `%vn_op` and `%vfs_op`. These meta-functions are listed in Table 7.

Vnode Meta-Function	VFS Meta-Function	Meaning
%vn_op_all	%vfs_op_all	all operations
%vn_op_construct	%vfs_op_construct	operations that create new vnodes
%vn_op_destroy	%vfs_op_destroy	operations that delete existing ones
%vn_op_read	%vfs_op_read	operations that read values
%vn_op_write	%vfs_op_write	operations that write values
%vn_op_pathname	%vfs_op_pathname	operations that manipulate path names
%vn_op_this	%vfs_op_this	The current operation being executed

Table 7: FiST Meta Functions

4.5.1 Errors

I chose to treat error codes as just another type of data. Error codes are usually a short integer: zero indicates no error, and a positive integer indicates one of many possible *errno* numbers. The directive `%error` returns the error code for the last function executed.

FiST can be instructed to return any error code which exists in `<sys/errno.h>` or even new error codes. New error codes would be suitable for new types of failure modes. For example, an encryption file system might have a new error code "invalid key;" a compression file system might have a code indicating "file already compressed;" a caching file system might have a code for "cached file is too old," and so on.

4.5.2 State Functions

A persistent file system (see Section 4.9.3) needs to store state in an auxiliary file system. The information stored needs to be formatted to fit off-line storage. For example, it must not contain pointers that may be valid in memory at the moment, but are certainly invalid after a reboot. In addition, facilities are needed for describing what information is stored in the state file system, and in what format it is stored.

State is defined by assigning a key and optional value to the FiST `%state` function. FiST knows when state needs to be actually flushed to permanent storage. It knows when you are assigning to it (writing state), or reading from it. The syntax for setting a state is as follows: `%state` *op, keylist, valuelist* where *op* is an operation to perform on the state. It can be one of `add`, `del`, `overwrite`, `addunique`, etc. I.e., normal data structure lookup table operations one might expect. The parameter *keylist* is a list of one or more keys and *valuelist* includes zero or more values.

The overall key is stored using a concatenation of the values of the keys; since no pointers are allowed, if any are specified, they would have to first be followed until their value is reached. No effort will be attempted to follow arbitrarily complex pointers.

The overall value stored is the list of concatenated values. Each value is preceded by the number of bytes stored. A valueless key has a zero byte count. The whole sequence is preceded by the number of items stored. If no values are specified, the number of items stored is set to zero.

The syntax for reading values from Statefs is as follows: `%state` *op, keylist* where *op* is `get`, and *keylist* is the same as when writing state. If the entry does not exist, the operation will return the error code ENOENT ("no such entry"). If it is empty, a non-zero integer will be returned. Otherwise, the list of values will be returned into the same type variables as were assigned to when the state was written.

4.6 Variables

FiST variables begin with the `$` character. The variable `$$` refers to the file system being defined; i.e., the one that interposes. If this file system interposes on top of more than one other file system, then those file systems may be referred to using the positional variables `$1`, `$2`, `$3`, etc. If only one file system is being interposed upon, then `$1` may be omitted.

The order for which a positional variable is assigned depends on the mounting options and the implementation. For example, when writing FiST code, using `$1` will refer to the first mounted file system on the command line, `$2` will refer to the second, etc. Changing which file system refers to which positional variable is as simple as mounting the file system with a different order of options.

To refer to a particular attribute of a vnode, the attribute keyword is appended to the positional parameter, separated by a period. For example:

- `$$.%type` refers to the type of vnode in this file system.
- `$2.%data` refers to data blocks of the second interposed file system.
- `$3.%error` refers to the error code returned from the third interposed file system.
- `$1.%mode` refers to mode bits of the first interposed file system.

4.7 Filters

A key idea in FiST is the use of filters. Filters are functions that act much like Unix filters — programs that may be concatenated with the shell pipe symbol `|` (a vertical bar). A filter accepts data as input, manipulates it, and then passes it on.

A filter is defined as follows: `%filter` *filtername fsindex attr* [{ *conditions* }] where *filtername* is the name of the filter; e.g., "gzip," "compress," "crypt," "DES," and "rot13." *fsindex* refers to the positional parameter of the file system such as `$$`, `$1`, and so on. *attr* refers to the attribute of the vnode, e.g. `%name`, `%owner`, `%mode`, or the vnode operation name such as `vn_read`, `vn_open`, etc.

An optional set of conditions may be supplied, enclosed in curly braces. If all the conditions are met, the filter will be applied. Conditions are separated by a semicolon. Each condition is a boolean expression using C syntax binary operators `&&`, `||`, `==`, `!=`, etc.

4.7.1 Filter Examples

Here are a few examples of filters.

1. To compress data blocks of regular files:

   ```
   %filter gzip $$ %data {$$.%type == regular}
   ```

2. To apply the Unix `crypt(3)`-based filter to all directories owned by user "ezk" (uid 2301) on the first file system:

   ```
   %filter crypt $1 %data {$1.%type == dir && $1.owner == 2301}
   ```

3. To expand shell environment variables that may be embedded in names of symbolic links:

   ```
   %filter envexpand $$ %name {$$.%type == link}
   ```

4. One may want to ignore errors returned by a caching file system, since data not in the cache can always be retrieved from the source. For example, if a file could not be written because the cache is full, that should not result in the vnode operation failing. To ignore out-of-space errors from the cache file system, one might use the "ignore" (null) filter:

   ```
   %filter ignore $2 %error {$2.%vn_op == write && $2.%error == ENOSPACE}
   ```

5. To log all attempts to read my directories by any non-system user other than the owner:

   ```
   %filter syslog $$ %vn_readdir {%cur_uid > 999 && %owner != %cur_uid}
   ```

4.7.2 Filter Functions

If the conditions of the filter are met, then a C function that implements the filter is called. The prototype of the function is as follows:

`int fist_filter_` *filtername*`_` *attr*`( & ` *attr-data-type*`, & ` *attr-data-size*`, ... );`

That is, the name of the filter function is composed from the filter name and the attribute type. The function receives at least two arguments: a pointer to the data that fits the type, and a pointer to the size of the data being passed.

Note that having the filter name and attribute type in the function's name could be easily done in C++ using methods and overloaded prototypes. This information is included in the function name because the code should be C, a requirement for portability.[8]

For example, for the first example in Section 4.7.1, the prototype would be:

[8] Most kernels (as well as Amd) were written in C, and cannot handle module linkage of objects that are written in C++. There are more C compilers available than C++ compilers, and C compilers generally produce faster and smaller object modules.

```
int fist_filter_gzip_data( page_t *, int * );
```

and for the third example it would be:

```
int fist_filter_envexpand_name( char **, int * );
```

Filter functions should behave like system calls, returning 0 (zero) upon success, and a non-zero integer if any failure occurred. Failure codes are assumed to be *errno* values.

To write a new filter, all one must do is write a simple C function that manipulates the data as needed. There is no need to worry about what vnode operations this would have to apply to, where the information is stored, when to allocate or free vnodes, most errors, and so on. All these are handled automatically by FiST.

4.8 Language Syntax

A FiST input file has four main sections, shown in Figure 14 with the appropriate delimiters:

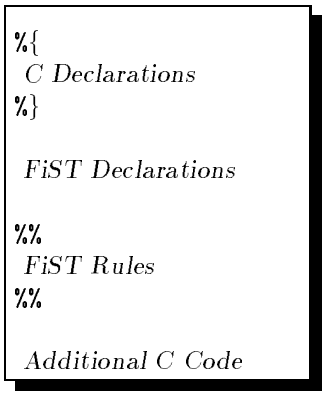

Figure 14: FiST Grammar Outline

Comments (enclosed in `/* ... */`) may appear in any of the sections and will be copied verbatim to the output file. C++ style comments starting with `//` are only useful for the FiST input file, as they get completely stripped during translation.

4.8.1 C Declarations

This optional section is copied verbatim to the head of the output file. This section is for `#include` statements, macros, forward and extern prototypes, etc., for functions whose code may exist later or outside the module.

4.8.2 FiST Declarations

The FiST declarations section contains keywords and other definitions that change the overall behavior of the produced code. They are listed in Table 8.

Keyword	Possible Values	Default Value	Comments
%fstype	stateless incore persistent	incore	Defines the type of file system to be generated, as described in Section 4.9. If the persistent file system type is chosen, an implicit additional file system is included to the number of interposers. The latter has a special index $0.
%interface	vnode, nfs	vnode	Defines the default interface to generate code for. Can also be defined or overriden by a command line option to the translator.
%interposers	*integer*	1	Defines how many file systems will this one directly access. If more than one, then to reference these file systems in order use the FiST variables $1, $2, $3, etc.
%mntopts	*struct* {...};	NULL	Defines a C structure with types and field names of arguments that need to be passed from the user process that mounts this file system, via the `mount(2)` system call, to the VFS mount operation. User level mount code and common header files will be generated for these definitions.
%filter	See Section 4.7	none	Defines FiST filter as described in Section 4.7

Table 8: FiST Declaration Keywords

If only one interposed file system is defined in the `%interposers` keyword in the declarations section, then its positional parameter may be omitted. All of the filter declarations described in Section 4.7.2 go in this section.

4.8.3 Rules

Filters are a construct that is useful when a "stream" of data needs to be modified the way Unix filters do. FiST Filters are just a specialization of the more general construct — Rules. FiST Rules allow finer and more flexible control over errors, arguments, and even data. Rules can access global data, where filters may not.

Rules for vnode operations take precedence over the filter definition of a vnode function. Each rule has the form of a FiST operation, followed by a single colon, optional action code, and terminated with a semicolon: *fistop*: *action* ; where *fistop* is a name of a vnode/VFS operation, optionally prefixed by a file system index variable and separated by a single dot. For example:

- `vfs_root` refers to the "get root vnode" VFS operation of the first and only interposed file system.

- `$1.vfs_root` refers to the same operation on the first interposed file system, when there are two or more interposed file systems.
- `$2.vn_getattr` refers to the vnode "get attributes" operation on the second interposed file system.
- `$0.vn_mkdir` refers to the vnode "make directory" operation on the state-storing interposed file system of a persistent file system.
- `$$.vn_setattr.error_action` refers to the error action code section of the vnode "set attributes operation of the current vnode. See Tables 9 and 10.

The optional action code, if included, must be delimited by a set of curly braces {...}. If the action is omitted, the default action is used. The pseudo-code for the default action for stateless and in-core FiST file systems is depicted in Figure 15, while pseudo-code for the default action for persistent file systems is shown in Figure 16.

FiST allows the file system designer to control each portion of the default code for stateless and in-core file systems. Keywords for each section are listed in Table 9.

```
 define variables --- optional
 manipulate the incoming arguments vector --- optional
foreach  f  in  all interposers of this file system
do
  error = $ f.fistop( args);
  if (error == ERROR); then
      perform actions based on errors --- optional
     return  error;
  endif
   manipulate the returning arguments vector --- optional
done
```

Figure 15: FiST Default Rule Action for Stateless and In-Core File Systems (Pseudo-Code)

Keyword	Code Section
%variables	define local variables
%in_args	manipulate the incoming arguments vector
%error_action	perform actions based on errors
%out_args	manipulate the returning arguments vector

Table 9: Code Section Names for Stateless and In-Core File Systems

FiST also lets the file system designer to control each portion of the default code for persistent file systems. Keywords for each section are listed in Table 10.

The code is treated as normal C code, but certain special variables and functions are interpreted and expanded at code generation time. The variables that are specially expanded are the positional variables $$, $0, $1, $2, $3, etc. Special functions that are available would include all filter functions defined above:

- functions to access the state file system (write state, read state, lookup state, etc.)
- functions to run a filter on data (de/compress, encrypt/decrypt)
- functions to manipulate pathnames (expand, translate)

```
 define variables --- optional
 lock this vnode and all vnodes in the interposition chain.
 manipulate the incoming arguments vector --- optional
foreach f in all interposers of this file system ; do
  retval[ f] = $ f.fistop( args);
   manipulate the returning arguments vector --- optional
done
if any error occurred ; then
   perform actions based on errors --- optional
   unlock interposition chain (and possibly unroll action).
  return error ;
endif
 save any state defined on $0.
 final manipulation of return codes
 unlock interposition chain.
return status-code ;
```

Figure 16: FiST Default Rule Action for Persistent File Systems (Pseudo-Code)

Keyword	**Code Section**
%variables	define local variables
%in_args	manipulate the incoming arguments vector
%action	retval[f] = $f.fistop(args);
%out_args	manipulate the returning arguments vector
%error_action	perform actions based on errors
%out_state	save any state defined
%out_error	final manipulation of return codes

Table 10: Code Section Names for Persistent File Systems

- functions to manipulate user credentials and file modes
- error handling when using more than one interposed file system (fail first, run through the end and return worst failure, restart operation on error, number of retries, etc.)

4.8.4 Additional C Code

This optional section is copied verbatim to the end of the output file. This section may contain the filter functions and any other auxiliary code.

4.9 File System Types

File systems generated by FiST may be classified depending on how they store their state, if any. File systems can have no state, regenerateable memory-resident state, or state that must be stored onto persistent media.

4.9.1 Stateless File Systems

A stateless file system does not create a "wrapping" vnode for every vnode in the interposed file system. As shown in Figure 17, there is only one new vnode created, as is needed for every file system: the root (Y2) of the interposing file system Y.

This file system type is quite limited. The only time that something interesting can happen is when the file system's mount point is crossed. I expect very few useful file systems to fall into this category. An example is Crossfs (Appendix B.1.2), a file system that performs a stateless event when a lookup operation traverses into it from the file system it is mounted on. A typical event might be to print a message on the system console that includes the uid and gid of process that crossed into this file system.

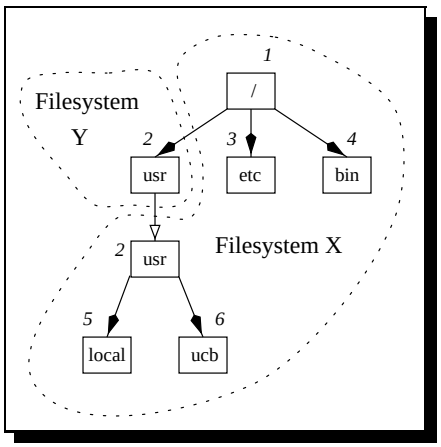

Figure 17: Vnode Structure in a Stateless File System

4.9.2 In-Core File Systems

An in-core file system is the type that has been developed so far in this proposal. State is maintained by the interposing vnodes.

The main attraction of an in-core file system is that its state may be regenerated after an unmount, reboot, or crash. In general, the state of the file system can be recovered by simply remounting it. A secondary advantage to in-core file systems is their simplicity. With just a few small modifications to Wrapfs one can generate many interesting and useful file systems, as exemplified in Appendix B.2. I expect many file systems generated by FiST to fall into this category.

4.9.3 Persistent File Systems

Persistent file systems require permanent state. To increase performance and fault tolerance, the state might typically be stored on a local disk, but remote file servers could be used just as well.

Figure 18 shows what happens when file system Y interposes on top of file system X. In that respect it is similar to an in-core file system. However, file system Y also uses an auxiliary file system W.

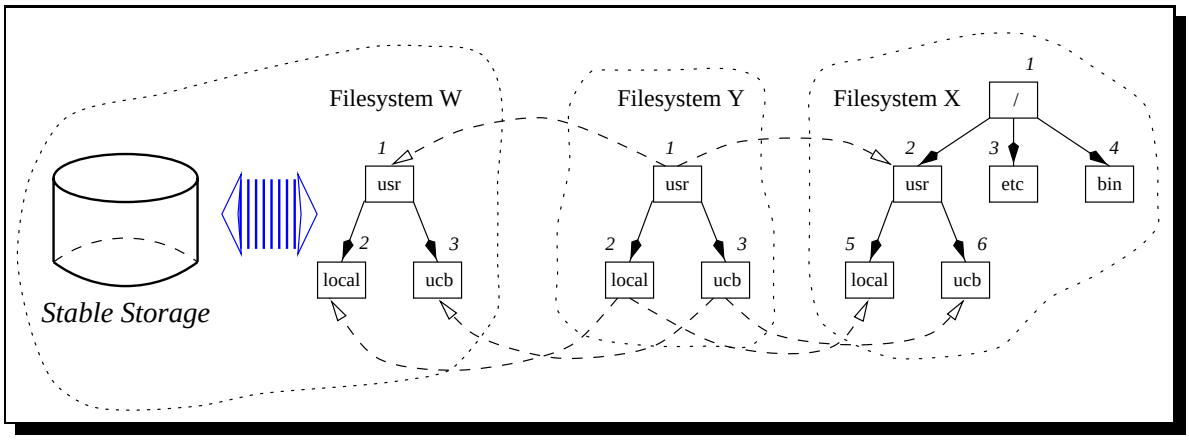

Figure 18: Vnode Structure in a Persistent File System

An example stateful file system is Cachefs (Appendix B.3.1), a file system that is used as a cache by another file system. When the file system is re-mounted the cache could be used again, subject to consistency constraints.

I require that operations on the state-storing file system be vnode operations. This has two benefits:

- The code is portable because it does not directly access native file systems.
- State may be stored on any type of media, since access is via the `VFS_*` and `VOP_*` macros (depicted in Appendices A.2 and A.4, respectively).

The restriction brings two disadvantages:

1. If only a little state is required, it could be stored in a much simpler data structure. Requiring state operations to go through all the file system layers may be unnecessarily costly.
2. The data structures representing the state may be too complex to be trivially stored in a Unix file system tree structure. Unix file systems offer a traditional tree-like organization. That makes storing state in such a data structure obvious, as there is a one-to-one mapping of source file to auxiliary state file. But what if the auxiliary state that needs to stored requires a more complex data structure, such as a B-tree [Elmasri94] or a graph? In that case, there is no simple way to take advantage of Unix's existing file system structures. Rather, the only way such state can be stored is *within* one or more "flat" Unix files, where an application level process will have to maintain the complex data structures within.

I think, however, that the benefits to my restriction outweigh the disadvantages. I intend to devise a state-storing file system using FiST, called Statefs. This file system — described in detail in Appendix B.2.3 — will do nothing more than record state for another file system. Statefs will operate inside the kernel, hopefully making its performance a non-issue.

For example, consider the Versionfs file system described in Appendix B.3.5. Unix file systems do not have versioning capabilities. If one wanted to add version information per file, without modifying the implementation *or* content of an existing file system, one would have to store the version information in an auxiliary location, and somehow correlate data in the unmodified file system with the auxiliary location. With FiST, one could create a file system that interposes onto two others: the unmodified data file system, and the auxiliary location. The latter can be any type of file system. FiST provides the facilities to make the necessary correlations between the "source" file system and the "auxiliary" one that is used for storing the extra versioning information. This auxiliary file system is Statefs.

Note that Statefs is an in-core file system. Although it requires storage for itself, the storage need never be interposed upon, and therefore is not considered "state" which would make Statefs a persistent file system.

4.10 User Level File Systems

User level NFS file system modules do not have the same functionality because the NFS protocol is more restrictive; the set of operations NFS provides is not as rich as the vnode/VFS set.

4.10.1 Types of User-Level FiST File Systems

The three different types of file systems — stateless, in-core, and persistent — can also be generated at the NFS level.

4.10.1.1 Stateless NFS File Systems

Stateless file systems perform interesting operations only when the mount point is crossed. Amd and all user-level file servers that I know of are contacted by the kernel only *after* the kernel has crossed their mount point during a lookup operation. Therefore, there is no logical place in a user-level automounter to call an operation when a mount point is crossed.

Since this type of file system is very limited in use and functionality, I will devote little or no effort to getting this case working in FiST.

4.10.1.2 In-Core NFS File Systems

This type of file system is the most natural to generate as a user level file system. In-core kernel-resident file systems keep state that can be regenerated — a vnode for every interposed file. An interesting observation is that inside NFS file systems, similar state is kept associated with an NFS file handle. That is exactly how Hlfsd and Amd are written: there exist `nfs_fhandle` structures for every file that is being represented by the file system.

4.10.1.3 Persistent NFS File Systems

Persistent file systems are more complicated than in-core file systems. However, it is much easier to

produce *user-level* code for persistent file systems. There is no longer a need for an auxiliary "state storing file system." Outside the kernel one may use any system call (like `read()` and `write()`), so state for example can be stored in any file on a local UFS disk.

4.10.2 Fan-Out in User-Level File Systems

Fan-out defines how many file systems one module can interpose upon. Inside the kernel there is no limit. The interposed vnode pointers are stored inside the interposer's private data field, then accessed as described in Section 3.3 and depicted in Figure 19.

```
/* perform FOO operation on two interposed vnodes */
int
vn_foo(vnode_t *vp, args) {
  vnode_t *hidden_vp1 = vp->v_data->hidden[0];
  vnode_t *hidden_vp2 = vp->v_data->hidden[1];
  int error;

  error = VN_FOO(hidden_vp1, args);

  if (error)
    return(error);

  error = VN_FOO(hidden_vp2, args);

  return(error);
}
```

Figure 19: Fan-Out in Stackable Vnode File Systems

This code is nice because it does not know about the type of the file systems it interposes upon. This is the result of *having* an abstract vnode interface in the first place. NFS is not an abstract interface like the vnode interface is. Therefore, an NFS module inside Amd would have to know what type of file system it is accessing:

- If the interposed file operation it is calling is another NFS module in that same Amd, just call that C function directly. This is just a simple optimization, but at the same time may avoid deadlocks when Amd may be waiting for an operation that needs to use the same Amd process.
- If the interposed file operation it is calling is not another NFS module in Amd, it would have to call standard system calls like `read()`, `write()`, `link()`, `mkdir()` etc.

The generated code must have some hooks that can probe an Amd server at run-time to see if the function it needs to call is local to the running process or not. This is a small complication to the generated code that may make it less clean. For example, the same vnode operation as in Figure 19, when generated for the NFS interface, would look much like the code in Figure 20.

Additional and detailed examples of using FiST are included in Appendix C.

```
/* perform FOO operation on two interposed nodes */
int
nfs_foo(fhandle_t *fhp, args)
{
  fhandle_t *hidden_fhp1 = fhp->fh_data->hidden[0];
  fhandle_t *hidden_fhp2 = fhp->fh_data->hidden[1];
  int error;

  /* find type of first handle, and call it */
  if (file_system_local_to_amd(fs_type_of(hidden_fhp1)))
    error = AMD_FOO(hidden_vhp1, args);
  else
    error = syscall(SYS_FOO, hidden_vhp1, args);

  if (error)
    return(error);

  /* find type of second handle, and call it */
  if (file_system_local_to_amd(fs_type_of(hidden_fhp2)))
    error = AMD_FOO(hidden_vhp2, args);
  else
    error = syscall(SYS_FOO, hidden_vhp2, args);

  return(error);
}
```

Figure 20: Fan-Out in Stackable NFS File Systems

5 Evaluation Plan

In evaluating FiST-produced file systems, it is important to keep in mind their purpose and "competition." First, the goals for stackable file systems include portability, ease of development, and ability to perform quick prototyping of new ideas. Therefore, some amount of performance degradation is expected and acceptable, given all the other benefits.

Second, "stacked" file systems should be compared to other interposing file systems, such as Sun's caching file system, Cachefs. It is inappropriate to compare interposing file systems to lower level file systems such as UFS or NFS, since the latter call device drivers directly without any additional overhead. Interposing file systems must incur some extra overhead because they must store and continually dereference information about the interposed file system(s).

On the other hand, it would be unfair to only compare kernel-resident interposing file systems to out of kernel file systems. Given all the context switches needed to communicate between the kernel and a user level server, it is not surprising that user level file systems are slower.

5.1 Criteria for Success

Given the above, these are the criteria I have set for testing the success of my work:

1. I should be able to generate at least one working, useful, and non-trivial file system in each of the categories of stateless, in-core, and persistent. I intend to generate the following FiST file systems: Crossfs (Appendix B.1.2), Cryptfs (Appendix B.2.6), and Cachefs (Appendix B.3.1).

2. For each such kernel level file system generated from a FiST description, I should be able to generate a user-level file system that runs in Amd.

3. The same FiST inputs should generate working file systems on at least three different Unix operating systems. I intend to produce code for Solaris (SVR4 based), FreeBSD (BSD-4.4-Lite based), and another operating system that has an established vnode interface, but is sufficiently different from "pure" SVR4 or BSD (for example HP-UX, AIX, or Digital Unix).

4. The overhead of interposition should be comparable to that of previous work on stackable file systems, and should not exceed 10% for Wrapfs. See Section 3.6.2 for details of current performance.

5. I should be able to show how to write FiST descriptions for a variety of other file systems.

5.2 Experiments

I intend to compare file systems in several categories:

- In-kernel file systems produced automatically using FiST against in-kernel hand written ones. For example Cachefs as described in Appendix B.3.1 and [SunSoft94].

- User-level file systems produced automatically using FiST against user-level hand written ones. For example Cryptfs as described in Appendix B.2.6 and [Blaze93].

- FiST generated file systems against another system that provides native stacking, such as UCLA's work [Heidemann94].
- Various FiST generated file systems vs. each other. For example an in-kernel Gzipfs (Appendix B.2.5) against a user-level one.

For each category, I will run the following tests:

1. Compare the performance of the file systems with similar or identical functionality.
2. Compare the size of the FiST input to the generated C code.
3. Compare the size of the FiST generated code to that of hand-written file systems (when sources for the latter are available).
4. Compare the effort required to write a file system using FiST vs. hand writing one (pending the availability of such information.)

Additionally I intend to find out how many different operating systems I can generate a file system for, from the same FiST input.

5.3 Lessons to be Learned

Lessons I expect to learn from this work include:

1. How easy or hard it is to use FiST to describe file systems at a high level — something that has never been done before.
2. The degree of portability of FiST-generated file systems across different platforms.
3. The performance of FiST-generated file systems compared to equivalent hand-written, optimized file systems.
4. The performance of *identical* file systems when run in-kernel versus at user level.
5. How difficult it is to write a file system from scratch versus describing it in FiST.

6 Related Work

Besides the previous efforts at vnode stacking mentioned in Section 2, there are several other approaches to providing flexible file systems.

6.1 HURD

The "Herd of Unix-Replacing Daemons" (HURD) from the Free Software Foundation (FSF) is a set of servers running on the Mach 3.0 microkernel that collectively provide a Unix-like environment. HURD file systems are implemented at user level, much the same as in Mach [Accetta86] and CHORUS [Abrosimov92].

The novel concept introduced by HURD is that of the translator. A translator is a program that can be attached to a pathname and perform specialized services when that pathname is accessed.

For example, in the HURD there is no need for the `ftp` program. Instead, a translator for ftp service is attached to a pathname, for example, `/ftp`. To access, say, the latest sources for the HURD itself, one could `cd` to the directory: `/ftp/prep.ai.mit.edu/pub/gnu` and copy the file `hurd-0.1.tar.gz`. Common Unix commands such as `ls`, `cp`, and `rm` work normally when applied to remote ftp-accessed files. The ftp translator takes care of logging into the remote server, translating FTP protocol commands to file system commands, and returning result codes back to the user.

Originally, a translator-like idea was used by the "Alex" work and allowed for example transparent ftp access via a file system interface [Cate92].

6.1.1 How to Write a Translator

HURD defines a common interface for translators. The operations in this interface are much closer to the user's view of a file than the kernel's, in many cases resembling Unix commands:

- `file_chown` to change owner and or group.
- `file_chflags` to change file flags.
- `file_utimes` to change access and modify times.
- `file_lock` to apply or manipulate advisory locks.
- `dir_lookup` to translate a pathname.
- `dir_mkdir` to create a new directory.

The HURD also includes a few operations not available in the vnode interface, but which have often been wished for:

- `file_notice_changes` to send notification when a file changes.
- `dir_notice_changes` to send notification when a directory changes.
- `file_getlinknode` to get the *other* names of a hard-linked file.

- `dir_mkfile` to create a new file without linking it into the file system. This is useful for temporary files, for preventing premature access to partially written files, and also for security reasons.
- `file_set_translator` to attach a translator to a point in the name space.

I have listed only some of the HURD file and directory operations, but even an exhaustive list is not as long as the vfs and vnode interfaces listed in Sections A.2 and A.4.

HURD comes with library implementations for disk-based and network-based translators. Users wishing to write new translators can link with `libdiskfs.a` or `libnetfs.a` respectively. If different semantics are desired, only those necessary functions must be modified and relinked. HURD also comes with `libtrivfs.a`, a trivial template library for file system translators, useful when one needs to write a complete translator from scratch.

6.1.2 Conclusions

The HURD is unlikely ever to include a "standard" vnode interface. For political and copyright reasons, the HURD was designed and built using free software and standards, with the emphasis on changing anything that could be improved. This undoubtedly will limit its popularity. That, coupled with the very different programming interface it offers, means that there is less need for something like a FiST translator to provide vnode-like code translation for the HURD. Nevertheless, the HURD offers an interface that is comparable to the vnode one and more.

6.2 Plan 9

Plan 9 was developed at Bell Labs in the late 1980's [Pike90, Pike91, Presotto93]. The Plan 9 approach to file system extension is similar to that of Unix.

The Plan 9 mount system call provides a file descriptor that can be a user process or remote file server. After a successful mount, operations below the mount point are sent to the file server. Plan 9's equivalent of the vnode interface (called 9P) comprises the following operations:

1. `nop`: The NULL ("ping") call. It could be used to synchronize a file descriptor between two entities.
2. `session`: Initialize a connection between a client and a server. This is similar to the VFS mount operation.
3. `attach`: Connect a user to a file server. Returns a new file descriptor for the root of the file system. Similar to the "get root" vnode operation.
4. `auth`: Authenticate a 9P connection.
5. `clone`: Duplicate an existing file descriptor between a user and a file server so that a new copy could be operated upon separately to provide user-specific name space.
6. `walk`: Traverse a file server (similar to lookup).
7. `clwalk`: Perform a clone operation followed by a walk operation. This one is an optimization of this common sequence of operations, for use with low-speed network connections.

8. `create`: Create a new file.
9. `open`: Prepare a file descriptor before read or write operations.
10. `read`: Read from a file descriptor.
11. `write`: Write to a file represented by a file descriptor.
12. `clunk`: Close a file descriptor (without affecting the file).
13. `remove`: Delete an existing file.
14. `stat`: Read the attributes of a file
15. `wstat`: Write attributes to a file.
16. `flush`: Abort a message and discard all remaining replies to it from a server.
17. `error`: Return an error code.

These operation messages are sent to a file server by the Plan 9 kernel in response to client requests, much the same way as user-level NFS servers behave.

My impression is that Plan 9 and 9P provide little benefit over what can be done with the vnode interface and a user level NFS server. Certainly, there is no major novelty in Plan 9 likes the translation concept of the HURD. Support for writing Plan 9 file servers is limited, and the functionality they can provide is not as well thought out as the HURD's. The HURD therefore provides a more flexible file service extension mechanism.

Changing FiST's language and translator to generate Plan 9 file system code would be no more difficult than doing it for the HURD.

6.2.1 Inferno

Inferno is Lucent Technologies' ("Bell Labs") successor to Plan 9. The Inferno network operating system was designed to be compact while fully functional, and fit in a small amount of memory. It is designed to run on devices such as set-top boxes, PDAs, and other embedded systems [Lucent97].

In Inferno, everything is represented by files. Therefore, file systems are indistinguishable from other services; they are all part of the Inferno name space. Even devices appear as small directories with a few files named "data," "ctl," "status," etc. To control an entity represented by such a directory, you write strings into the "ctl" file; to get status, read the "status" file; and to write data, open the "data" file and write to it. This model is simple and powerful: operations can be done using simple open, read/write, and close sequences — all without the need for different APIs for networking, file systems, or other daemons [Breitstein97].

Inferno allows name spaces to be customized by a client, server, or any application. The *mount* operation imports a remote name space onto a local point, much like Unix file system mounts work. The *bind* operation is used to make a name space in one directory appear in another. This is similar to creating symbolic links and hard links in traditional Unix file systems, with the exception that Inferno can also unify the contents of two directories.

For Inferno to offer a new file system functionality that might otherwise be achieved via vnode stacking, an application has to *mount* and *bind* the right name spaces, add its own as required (implemented via the Limbo programming language [Kernighan96]), and then offer them for importation (which can be done securely).

Inferno's main disadvantage is a familiar one. It is a brand new operating system, and employs a new programming language and model. Inferno is not likely to be as portable and in wide use for years to come. My impression of Inferno is that if successful, it will become popular in the field of embedded systems.

6.3 Programmed Logic Corp.'s StackFS

Programmed Logic Corp. is a company specializing in storage products. Among their offerings are a compression file system, a 64-bit file system, a high-throughput file system utilizing transactions, and a stackable file system. PLC's StackFS [PLC96] is very similar to my wrapper file system described in Appendix B.2.1.

StackFS allows for different modules to be plugged in a variety of ways to provide new functionality. Modules offering 64-bit access, mirroring, union, hierarchical storage management (HSM), FTP, Caching, and others are available. Several modules can be loaded in a stack fashion into StackFS. The only organization available is a single stack; that is, each file system performs its task and then passes on the vnode operation to the one it stacked on top of, until the lowest stacked file system access the native file system (UFS or NFS).

There is no support for fan-in or fanout. There is seemingly no support for composition either. Also, StackFS does not have facilities for saving state in an auxiliary file system the way FiST defines Statefs (see Appendix B.2.3). Finally, there is no language available for producing modules that will work within StackFS. Still, PLC's products are the only known commercially available stackable file system implementation.

6.4 Spring

Spring is an object oriented research operating system built by Sun Microsystems Laboratories [Mitchel94]. It was designed as a set of cooperating servers on top of a microkernel. Spring uses a modified Interface Definition Language (IDL) [Stone87, Warren87] as outlined in the CORBA specifications [CORBA91] to define the interfaces between the different servers.

Spring includes several *generic* modules that provide services that are useful for file systems:

- **Caching**: A module that provides attribute caching of objects.
- **Coherency**: A layer that guarantees object states in different servers are identical. It is implemented at the page level, so that every object inherited from it could get coherency "for free."
- **I/O**: A layer that lets one perform streaming-based operation on objects such as used by the Unix `read` and `write` system calls.
- **Memory Mapper**: A module that provides page-based caching, sharing, and access (similar to the Unix `mmap` system call, and more).

- **Naming**: A module that maintains names of objects.

- **Security**: A module that provides secure access and credentials verification of objects.

Spring file systems inherits from many of the above modules. The naming module provides naming of otherwise anonymous file objects, giving them persistence. The I/O layer is used when the `read` or `write` system calls are invoked. The memory pager is used when a page needs to be shared or when system calls equivalent of `mmap` are invoked. The security layer ensures that only permitted users can access files locally or remotely, and so on.

Spring file servers can reside anywhere — not just on the local machine or remotely, but also in kernel mode or in user-level. File servers can replace, overload, and augment operations they inherit from one or more file servers. This form of object oriented composition makes file systems simpler to write.

File system stacking is easy and flexible in Spring. The implementation of the new file system chooses which file system modules to inherit operations from, then changes only those that need modification. Since each file object is named, Spring stackable file systems can perform operations on a per-file basis; they can, for example, decide to alter the behavior of some files, while letting others pass through unchanged.

Spring is a research operating system used by Sun to develop new technology that could then be incorporated into its commercial operating system products. As such, performance is a major concern in Spring. Performance had always been a problem in microkernel architectures due to the numerous messages that must be sent between the many servers that could be distributed over distinct machines and even wide-area networks. Spring's main solution to this problem was the abundant use of caching. Everything that can be cached is cached: pages, names, data, attributes, credentials, etc. — on both clients and servers.

Without caching, performance degradation for a single stack layer file system in Spring ranged from 23%-39%, and peaked at 69%-101% for a two-layer stack (for the `fstat` and `open` operations)! With caching it was barely noticeable. However, even with caching extensively employed, basic file system operations (without stacking) still took on average 2-7 *times* longer than the highly optimized SunOS 4.1.3 [Khalidi93]. So while it is clear that caching helped to alleviate some overheads, many more remain. Compare that to FiST's *total* overhead for a single stack layer of about 3-6% (Section 3.6.2) and you see that FiST is more capable of commercial grade performance.

To implement a new stackable file system in Spring, one has to write only those operations that need implemented. The rest get their implementation inherited from other file system modules. FiST also lets you implement only those file system operations that are needed. Every operation you do not explicitly modify or override defaults to that of Wrapfs (forward the vnode operation to the interposed file system).

The work done in the Spring project is clean and impressive. Spring, however, still uses a different file system interface and as a research operating system is not likely to become popular any time soon, if ever. There is still plenty of merit to using FiST to provide as many of the file system facilities that Spring provides, using a simple to define language and generating code for a more common interface.

6.5 4.4 BSD's nullfs

4.4 BSD includes a file system called "nullfs" that is identical to my Wrapfs. BSD's nullfs does not create any infrastructure for stacking; all it does is allow mounting one part of the file system in a different location. It proved useful as a template from which 4.4 BSD's Union file system was written [Pendry95]. The latter was developed by extending nullfs to *merge* the mount point file system and the mounted one, rather than blindly forward vnode and VFS operations to the new mount point.

The only contribution of 4.4 BSD to stacking is that it used an existing vnode interface in a manner similar to FiST. In fact, the way to write stackable file systems in 4.4 BSD is to take the template code for their nullfs, and adapt it to one's needs.

7 Summary

The proposed work strives for a radical improvement in the ease and flexibility with which new file systems can be written and deployed. I expect the most significant contributions of my thesis to be:

1. The first language for the abstract description of file system behavior.
2. The first method for writing file systems without access to the sources for the target operating system.
3. The first method for writing file systems that are portable across different operating systems.
4. A mechanism to produce either kernel or user-level file systems from the same higher-level description.
5. The performance degradation added by my mechanism would be small.

A Appendix: Vnode Interface Tutorial

This section provides a simple introduction to the vnode interface. The information herein is gathered from pivotal papers on the subject [Kleiman86, Rosenthal90] and from system C header files — specifically `<sys/vfs.h>` and `<sys/vnode.h>`.

The two important data structures used in the vnode interface are `struct vfs` and `struct vnode`, depicted in Figures 21 and 25, respectively.

A.1 `struct vfs`

An instance of the `vfs` structure exists in a running kernel for each mounted file system. All of these instances are chained together in a singly-linked list. The head of the list is a global variable called `root_vp`, which contains the vfs for the root device. The field `vfs_next` links one vfs structure to the following one in the list.

```
typedef struct vfs {
  struct vfs      *vfs_next;              /* next VFS in VFS list */
  struct vfsops   *vfs_op;                /* operations on VFS */
  struct vnode    *vfs_vnodecovered;      /* vnode mounted on */
  u_long          vfs_flag;               /* flags */
  u_long          vfs_bsize;              /* native block size */
  int             vfs_fstype;             /* file system type index */
  fsid_t          vfs_fsid;               /* file system id */
  caddr_t         vfs_data;               /* private data */
  dev_t           vfs_dev;                /* device of mounted VFS */
  u_long          vfs_bcount;             /* I/O count (accounting) */
  u_short         vfs_nsubmounts;         /* immediate sub-mount count */
  struct vfs      *vfs_list;              /* sync list pointer */
  struct vfs      *vfs_hash;              /* hash list pointer */
  kmutex_t        vfs_reflock;            /* mount/unmount/sync lock */
} vfs_t;
```

Figure 21: SunOS 5.x VFS Interface

The fields relevant to this proposal are as follows:

- `vfs_next` is a pointer to the next vfs in the linked list.
- `vfs_op` is a pointer to a function-pointer table. That is, this `vfs_op` can hold pointers to UFS functions, NFS, PCFS, HSFS, etc. For example, if the vnode interface calls the function to mount the file system, it will call whatever subfield of `struct vfsops` (See Section A.2) is designated for the mount function. That is how the transition from the vnode level to a file system-specific level is made.
- `vfs_vnodecovered` is the vnode on which this file system is mounted (the mount point).

- `vfs_flag` contains bit flags for characteristics such as whether this file system is mounted read-only, if the setuid/setgid bits should be turned off when exec-ing a new process, if sub-mounts are allowed, etc.
- `vfs_data` is a pointer to opaque data specific to this vfs and the type of file system this one is. For an NFS vfs, this would be a pointer to `struct mntinfo` (located in `<nfs/nfs_clnt.h>`) — a large NFS-specific structure containing such information as the NFS mount options, NFS read and write sizes, host name, attribute cache limits, whether the remote server is down or not, and more.
- `vfs_reflock` is a mutual exclusion variable used by locking functions that need to change values of certain fields in the vfs structure.

A.2 `struct vfsops`

The vfs operations structure (`struct vfsops`, seen in Figure 22) is constant for each type of file system. For every instance of a file system, the vfs field `vfs_op` is set to the pointer of the operations vector of the underlying file system.

```
typedef struct vfsops {
  int     (*vfs_mount)();
  int     (*vfs_unmount)();
  int     (*vfs_root)();
  int     (*vfs_statvfs)();
  int     (*vfs_sync)();
  int     (*vfs_vget)();
  int     (*vfs_mountroot)();
  int     (*vfs_swapvp)();
} vfsops_t;
```

Figure 22: SunOS 5.x VFS Operations Interface

Each field of the structure is assigned a pointer to a function that implements a particular operation for the file system in question:

- `vfs_mount` is the function to mount a file system on a particular vnode. It is responsible for initializing data structures, and filling in the vfs structure with all the relevant information (such as the `vfs_data` field).
- `vfs_unmount` is the function to release this file system, or unmount it. It is the one, for example, responsible for detecting that a file system has still opened resources that cannot be released, and for returning an *errno* code that results in the user process getting a "device busy" error.
- `vfs_root` will return the root vnode of this file system. Each file system has a root vnode from which traversal to all other vnodes in the file system is enabled. This vnode usually is hand crafted (via `kernel_malloc`) and not created as part of the standard ways of creating new vnodes (i.e. `vn_lookup`).

- `vfs_statvfs` is used by programs such `df` to return the resource usage status of this file system (number of used/free blocks/inodes).
- `vfs_sync` is called successively in every file system when the `sync(2)` system call is invoked, to flush in-memory buffers onto persistent media.
- `vfs_vget` turns a unique file identifier *fid* for a vnode into the vnode representing this file. This call works in conjunction with the vnode operation `vop_fid`, described in Appendix section A.4.
- `vfs_mountroot` is used to mount this file system as the root (first) file system on this host. It is different from `vfs_mount` because it is the first one, and therefore many resources such as `root_vp` do not yet exist. This function has to manually create and initialize all of these resources.
- `vfs_swapvp` returns a vnode specific to a particular device onto which the system can swap. It is used for example when adding a file as a virtual swap device via the `swap -a` command [SMCC94b].

The VFS operations get invoked transparently via macros that dereference the operations vector's field for that operation, and pass along the vfs and the arguments it needs. Each VFS operation has a macro associated with it, located in `<sys/vfs.h>`. Figure 23 shows the definitions for these macros.

```
#define VFS_MOUNT(vfsp, mvp, uap, cr) (*(vfsp)->vfs_op->vfs_mount)(vfsp, mvp, uap, cr)
#define VFS_UNMOUNT(vfsp, cr)         (*(vfsp)->vfs_op->vfs_unmount)(vfsp, cr)
#define VFS_ROOT(vfsp, vpp)           (*(vfsp)->vfs_op->vfs_root)(vfsp, vpp)
#define VFS_STATVFS(vfsp, sp)         (*(vfsp)->vfs_op->vfs_statvfs)(vfsp, sp)
#define VFS_SYNC(vfsp, flag, cr)      (*(vfsp)->vfs_op->vfs_sync)(vfsp, flag, cr)
#define VFS_VGET(vfsp, vpp, fidp)     (*(vfsp)->vfs_op->vfs_vget)(vfsp, vpp, fidp)
#define VFS_MOUNTROOT(vfsp, init)     (*(vfsp)->vfs_op->vfs_mountroot)(vfsp, init)
#define VFS_SWAPVP(vfsp, vpp, nm)     (*(vfsp)->vfs_op->vfs_swapvp)(vfsp, vpp, nm)
```

Figure 23: VFS Macros

When any piece of file system code, that has a handle on a vfs, wants to call a vfs operation on that vfs, they simply dereference the macro, as depicted in Figure 24.

```
int foo(const vfs_t *vfsp, vnode_t **vpp)
{
  int error;

  error = VFS_ROOT(vfsp, vpp);
  if (error)
    return (error);
}
```

Figure 24: VFS Macros Usage Example

A.3 `struct vnode`

An instance of `struct vnode` (Figure 25) exists in a running system for every opened (in-use) file, directory, symbolic-link, hard-link, block or character device, a socket, a Unix pipe, etc.

```
typedef struct vnode {
  kmutex_t        v_lock;                 /* protects vnode fields */
  u_short         v_flag;                 /* vnode flags (see below) */
  u_long          v_count;                /* reference count */
  struct vfs      *v_vfsmountedhere;      /* ptr to vfs mounted here */
  struct vnodeops *v_op;                  /* vnode operations */
  struct vfs      *v_vfsp;                /* ptr to containing VFS */
  struct stdata   *v_stream;              /* associated stream */
  struct page     *v_pages;               /* vnode pages list */
  enum vtype      v_type;                 /* vnode type */
  dev_t           v_rdev;                 /* device (VCHR, VBLK) */
  caddr_t         v_data;                 /* private data for fs */
  struct filock   *v_filocks;             /* ptr to filock list */
  kcondvar_t      v_cv;                   /* synchronize locking */
} vnode_t;
```

Figure 25: SunOS 5.x Vnode Interface

Structure fields relevant to our work are:

- `v_lock` is a mutual exclusion variable used by locking functions that need to perform changes to values of certain fields in the vnode structure.
- `v_flag` contains bit flags for characteristics such as whether this vnode is the root of its file system, if it has a shared or exclusive lock, whether pages should be cached, if it is a swap device, etc.
- `v_count` is incremented each time a new process opens the same vnode.
- `v_vfsmountedhere`, if non-null, contains a pointer to the vfs that is mounted on this vnode. This vnode thus is a directory that is a mount point for a mounted file system.
- `v_op` is a pointer to a function-pointer table. That is, this `v_op` can hold pointers to UFS functions, NFS, PCFS, HSFS, etc. For example, if the vnode interface calls the function to open a file, it will call whatever subfield of `struct vnodeops` (See Section A.4) is designated for the open function. That is how the transition from the vnode level to a file system-specific level is made.
- `v_vfsp` is a pointer to the vfs that this vnode belongs to. If the value of the field `v_vfsmountedhere` is non-null, it is also said that `v_vfsp` is the parent file system of the one mounted here.
- `v_type` is used to distinguish between a regular file, a directory, a symbolic link, a block/character device, a socket, a Unix pipe (fifo), etc.

- `v_data` is a pointer to opaque data specific to this vnode. For an NFS vfs, this might be a pointer to `struct rnode` (located in `<nfs/rnode.h>`) — a remote file system-specific structure containing such information as the file-handle, owner, user credentials, file size (from the client's view), and more.

A.4 `struct vnodeops`

An instance of the vnode operations structure (`struct vnodeops`, listed in Figure 26) exists for each different type of file system. For each vnode, the vnode field `v_op` is set to the pointer of the operations vector of the underlying file system.

```
typedef struct vnodeops {
  int     (*vop_open)();          int     (*vop_fid)();
  int     (*vop_close)();         void    (*vop_rwlock)();
  int     (*vop_read)();          void    (*vop_rwunlock)();
  int     (*vop_write)();         int     (*vop_seek)();
  int     (*vop_ioctl)();         int     (*vop_cmp)();
  int     (*vop_setfl)();         int     (*vop_frlock)();
  int     (*vop_getattr)();       int     (*vop_space)();
  int     (*vop_setattr)();       int     (*vop_realvp)();
  int     (*vop_access)();        int     (*vop_getpage)();
  int     (*vop_lookup)();        int     (*vop_putpage)();
  int     (*vop_create)();        int     (*vop_map)();
  int     (*vop_remove)();        int     (*vop_addmap)();
  int     (*vop_link)();          int     (*vop_delmap)();
  int     (*vop_rename)();        int     (*vop_poll)();
  int     (*vop_mkdir)();         int     (*vop_dump)();
  int     (*vop_rmdir)();         int     (*vop_pathconf)();
  int     (*vop_readdir)();       int     (*vop_pageio)();
  int     (*vop_symlink)();       int     (*vop_dumpctl)();
  int     (*vop_readlink)();      void    (*vop_dispose)();
  int     (*vop_fsync)();         int     (*vop_setsecattr)();
  void    (*vop_inactive)();      int     (*vop_getsecattr)();
                                } vnodeops_t;
```

Figure 26: SunOS 5.x Vnode Operations Interface

Each field of the structure is assigned a pointer to a function that implements a particular operation on the file system in question:

- `vop_open` opens the requested file and returns a new vnode for it.
- `vop_close` closes a file.
- `vop_read` reads data from the opened file.
- `vop_write` writes data to the file.

- `vop_ioctl` performs miscellaneous I/O control operations on the file, such as setting non-blocking I/O access.
- `vop_setfl` is used to set arbitrary file flags.
- `vop_getattr` gets the attributes of a file, such as the mode bits, user and group ownership, etc.
- `vop_setattr` sets the attributes of a file.
- `vop_access` checks to see if a particular user, given the user's credentials, is allowed to access a file.
- `vop_lookup` looks up a directory for a file name. If found, a new vnode is returned.
- `vop_create` creates a new file.
- `vop_remove` removes a file from the file system.
- `vop_link` makes a hard-link to an existing file.
- `vop_rename` renames a file.
- `vop_mkdir` makes a new directory.
- `vop_rmdir` removes an existing directory.
- `vop_readdir` reads a directory for entries within.
- `vop_symlink` creates a symbolic-link to a file.
- `vop_readlink` reads the value of a symbolic link, that is, what the link points to.
- `vop_fsync` writes out all cached information for a file.
- `vop_inactive` signifies to the vnode layer that this file is no longer in use, that all its references had been released, and that it can now we deallocated.
- `vop_fid` returns a unique file identifier *fid* for a vnode. This call works in conjunction with the vfs operation `vfs_vget` described in Appendix section A.2.
- `vop_rwlock` locks a file before attempting to read from or write to it.
- `vop_rwunlock` unlocks a file after having read from or wrote to it.
- `vop_seek` sets the read/write head to a particular point within a file, so the next read/write call can work from that location in the file.
- `vop_cmp` compares two vnodes and returns true/false.
- `vop_frlock` perform file and record locking on a file.
- `vop_space` frees any storage space associated with this file.

- `vop_realvp` for certain file systems, returns the "real" vnode. This is useful in stackable vnodes, where a higher layer may request the real/hidden vnode underneath, so it can operate on it.
- `vop_getpage` reads a page of a memory-mapped file.
- `vop_putpage` writes to a page of a memory-mapped file.
- `vop_map` maps a file into memory. See [Gingell87a, Gingell87b] for more details.
- `vop_addmap` adds more pages to a memory-mapped file.
- `vop_delmap` removes some pages from a memory-mapped file.
- `vop_poll` polls for events on the file. This is mostly useful when the vnode is of type "socket" or "fifo," and replaces the older `vop_select` vnode operation. This operation is often used to implement the `select(2)` system call.
- `vop_dump` dumps the state of the kernel (memory buffers, tables, variables, registers, etc.) to a given vnode, usually a swap-device. This is used as the last action performed when a kernel panics and needs to save state for post-mortem recovery by tools such as `crash` [SMCC95].
- `vop_pathconf` supports the POSIX path configuration standard. This call returns various configurable file or directory variables.
- `vop_pageio` performs I/O directly on mapped pages of a file.
- `vop_dumpctl` works in conjunction with `vop_dump`. It is used to prepare a file system before a dump operation by storing data structures that might otherwise get corrupted shortly after a panic had occurred, and deallocates these private dump data structures after a successful dump.
- `vop_dispose` removes a mapped page from memory.
- `vop_setsecattr` is used to set Access Control Lists (ACLs) on a file.
- `vop_getsecattr` is used to retrieve the ACLs of a file.

Vnode operations get invoked transparently via macros that dereference the operations vector's field for that operation, and pass along the vnode and the arguments it needs. Each vnode operation has a macro associated with it, located in `<sys/vnode.h>`. Figure 27 shows as an example, the definitions for some of these calls.

When any piece of file system code, that has a handle on a vnode, wants to call a vnode operation on it, it simply dereferences the macro, as depicted in Figure 28.

```
#define VOP_OPEN(vpp, mode, cr)        (*(*(vpp))->v_op->vop_open)(vpp, mode, cr)
#define VOP_CLOSE(vp, f, c, o, cr)     (*(vp)->v_op->vop_close)(vp, f, c, o, cr)
#define VOP_READ(vp, uiop, iof, cr)    (*(vp)->v_op->vop_read)(vp, uiop, iof, cr)
#define VOP_MKDIR(dp, p, vap, vpp, cr) (*(dp)->v_op->vop_mkdir)(dp, p, vap, vpp, cr)
#define VOP_GETATTR(vp, vap, f, cr)    (*(vp)->v_op->vop_getattr)(vp, vap, f, cr)
#define VOP_LOOKUP(vp, cp, vpp, pnp, f, rdir, cr) \
                           (*(vp)->v_op->vop_lookup)(vp, cp, vpp, pnp, f, rdir, cr)
#define VOP_CREATE(dvp, p, vap, ex, mode, vpp, cr) \
                         (*(dvp)->v_op->vop_create)(dvp, p, vap, ex, mode, vpp, cr)
```

Figure 27: Some Vnode Macros

```
int foo(vnode_t *dp, char *name,
        vattr_t *vap, vnode_t **vpp, cred_t *cr)
{
  int error;

  error = VOP_MKDIR(dp, name, vap, vpp, cr);
  if (error)
    return (error);
}
```

Figure 28: Vnode Macros Usage Example

A.5 How It All Fits

To see how it all fits in, the following example depicts what happens when a remote (NFS) file system is mounted onto a local (UFS) file system, and the sequence of operations that a user level process goes through to satisfy a simple read of a file on the mounted file system.

A.5.1 Mounting

Consider first the two file systems X and Y, depicted in Figure 29. In this figure, the numbers near the node names represent the file/inode/vnode numbers of that file or directory within that particular file system. For example "X5" refers to the vnode of the directory `/usr/local` on file system X.

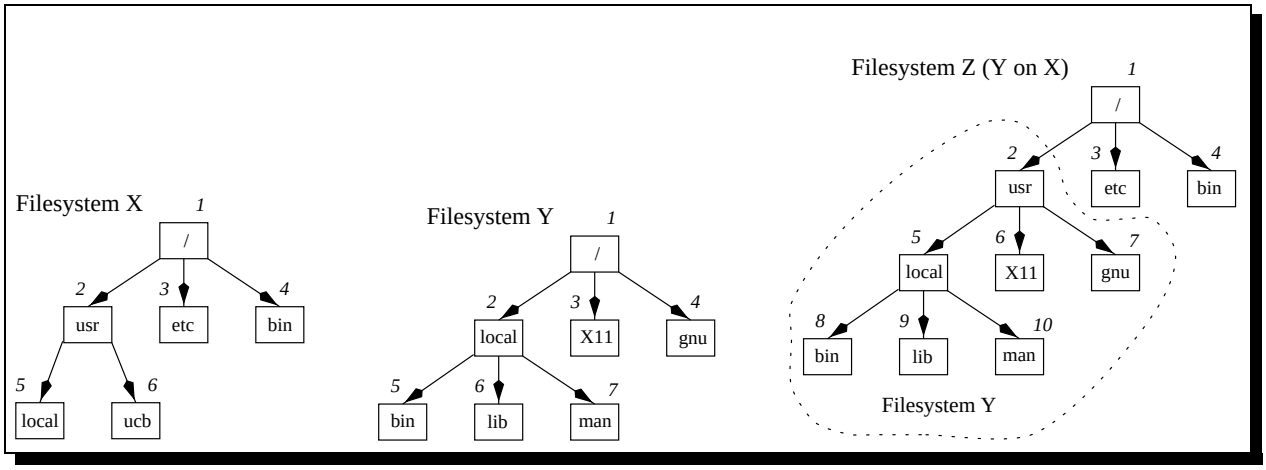

Figure 29: File System Z as Y mounted on X

Let's also assume that X is a UFS (local) file system, and that Y is the `/usr` file system available on a remote file server named "titan." We wish to perform the following NFS mount action: `mount titan:/usr /usr`.

The in-kernel actions that proceed, assuming that all export and mount permissions are successful, are the following:

1. A new vfs is created and is passed on to `nfs_mount`.
2. `nfs_mount` fills in the new vfs structure with the vfs operations structure for NFS, and sets the `v_vfsmountedhere` of the vnode X2 to this new vfs.
3. `nfs_mount` also creates a new vnode to serve as the root vnode of the Y file system as mounted on X. It stores this vnode in the `v_data` field of the new vfs structure.

A.5.2 Path Traversal

Figure 29 also shows the new structure of file system X, after Y had been mounted, as file system Z.

The sequence of in-kernel operations to, say, read the file `/usr/local/bin/tex` would be as follows:

1. The system call `read()` is executed. It begins by looking up the file.
2. The generic lookup function performs a `VOP_LOOKUP(rootvp, "usr")`. It tries to look for the next component in the path, starting from the current lookup directory (root vnode).
3. The lookup function is translated into `ufs_lookup`. The vnode X2 is found. Note that X2 is *not* the same vnode as Z2! X2 is hidden, while Z2 overshadows it.
4. The lookup function now notices that X2's `v_vfsmountedhere` field is non-null, so it knows that X2 is a mount point. It calls the `VOP_ROOT` function on the vfs that is "mounted here," that translates to `nfs_lookup`. This function returns the root vnode of the Y file system as it is mounted on X. This root vnode is X2. The "magic" part that happens at this point is that the lookup routine now resumes its path traversal but on the *mounted* file system.
5. An `nfs_lookup` is performed on the Z2 vnode for the component `"local"`, that will return the vnode Z5.
6. An NFS lookup is performed on vnode Z5 for the component `"bin"`, that will return the vnode Z8.
7. An NFS lookup is performed on vnode Z8 for the component `"tex"`, that will return the vnode for the file.
8. The lookup is complete and returns the newly found vnode for component `"tex"` to the `read()` system call.
9. The generic read function performs a `VOP_READ` on the newly found vnode. Since that vnode is an NFS one, the read is translated into `nfs_read`.
10. Actual reading of the file `/usr/local/bin/tex` begins in earnest.

B Appendix: Typical Stackable File Systems

This section lists a few typical file systems that can be generated using FiST. It is intended as a non-exhaustive listing of exemplary file systems that could be produced, so that references to them by name from other sections in this proposal could be made.

The file systems are classified into three categories as described in Section 4.9: stateless, in-core, and persistent.

B.1 Stateless File Systems

In a stateless file system, state is not required for the file system anywhere — neither in memory nor on disk. This means that the file system does not have to maintain vnode states. It does not need to create a "wrapping" vnode for every vnode in the interposed file system.

B.1.1 Nullfs

A file system that does nothing but pass the vnode operation to the underlying vnode. Not very useful beyond an educational exercise. The only interesting action that may occur happens when a mount point is crossed into Nullfs.

B.1.2 Crossfs

A file system that performs a simple event when a pathname lookup has traversed into it from the one it is mounted on. A typical event might be to print a message on the system console that includes the uid and gid of process that crossed into this file system. It is a simpler form of the Snoopfs file system (see Appendix section B.2.4).

B.2 In-Core File Systems

In an in-core file system, state for the file system is maintained only within the kernel's memory. The file system needs to create its own vnodes on top of lower level file systems. For each in-core vnode of the interposed file system, there will be a vnode in the interposer's file system. However, if the machine crashes and all contents of memory are lost, no permanent disk corruption would occur due to this file system's state not having been written out.

B.2.1 Wrapfs

Wrapfs is a template file system. It maintains a vnode for every open vnode on the interposed file system, and passes on the vnode or vfs operation to the interposed vnode, receiving its return status, and returning it back to the caller.

B.2.2 Envfs

A file system that expands some environment variables in path names. Envfs needs the list of variables and their values to expand, given to it as mount options. Envfs is very similar to Wrapfs. The only operation that is different in Envfs is `vn_lookup()`. All it has to do is expand any variable names to

their values within the interposing file system, modify the pathname component being looked up as needed, and then call the interposed file system.

Incidentally, that is not what I call "state," since it can be reproduced by remounting the file system with the same options. The state that is required is a vnode in Envfs for each vnode in the underlying file system. The reason we need it is so that open files in the interposing file system can refer to the proper interposed vnodes. For example, the current working directory (cwd) of Unix shells, is actually represented by an open directory vnode in the kernel. When a lookup operation occurs in Envfs, it starts from the directory vnode of the current working directory of the process in question; that is the vnode the kernel passes on to the lookup routine, and that operation must be able to access the interposed vnode for the lookup to proceed.

B.2.3 Statefs

A file system that will record a few pre-determined data structures in one or more files of their file system. Initially it will provide a simple lookup table functions that could be used once a state file has been read into memory. Later on it could be expanded to more complex and exotic off-line data structures such as B-trees [Elmasri94].

Since files in this file system will be completely under the control of the file system, it could be made hidden from users. User processes would not need to be able to modify these files. However, it would be useful for users to be able to list and read them for logging, reporting, and debugging purposes. In other words, it may be a read-only file systems as far as user-processes are concerned.

Statefs itself cannot be directly interposed upon. It can only be accessed within the implementation of another interposeable module (via $0, as described in Table 8). Statefs sole existence is to *augment* an existing file system's functionality, not to be the functionality itself. Therefore, at the moment, I see no reason to allow Statefs to be directly interposed upon.

B.2.4 Snoopfs

A file system that will tell you who accessed what files or directories, and when. The file system will record, via direct console messages or syslog [SMCC90], the uid and gid of a process accessing files in this file system, the names of the files or directories, and the time of access. After recording this information, Snoopfs will forward the vnode request to the interposed file system, thus hiding the fact that this file system is being monitored.

Unix file permissions provide a mechanism to protect one's files from prying eyes, but there are many ways for remote users, especially ones with local root access on their workstations, to become a different user (using the su program) and then try and access someone else's files. Besides, even if the user was unsuccessful at poking about someone else's files (maybe a student looking for a leftover copy of a final exam in their instructor's account), the fact that such access was attempted may be an interesting fact on its own.

B.2.5 Gzipfs

A compression file system using the GNU zip algorithms. Only file data should be compressed for performance reasons. File name extensions will be used to find files that are already compressed and avoid re-compressing them (a process that normally results in the *growth* of the file size). Data blocks

will be compressed before written out to the interposed's file system, and decompressed after being read from it and before being returned to the caller of the interposer's file system.

A slight modification of this file system would only compress files older than a certain date, thus allowing frequently accessible files speeder access, and saving space on seldom-used files.

The difficulty in implementing this file system will come from having to deal with the fact that the sizes of the data streams change when reading compressed files (size grows) and writing (size generally shrinks), and how to maintain the uncompressed file size while saving partition disk blocks resulting from files having been compressed.

B.2.6 Cryptfs

An encryption file system that will use similar algorithms as cfs [Blaze93]. For security reasons, all data blocks will be encrypted (both directory and file blocks). Data streams get encrypted before written to the interposed file system (on the way "down"), and decrypted after being read (on the way "up").

An added difficultly in writing this file system, in addition to the problems of stream size changes, will be key management. The file system should enable each individual user to have their own private key for decoding their own files within the encrypted file system.

B.2.7 Statsfs

A file system that will record statistics on the interposed file system, and report them via console messages or syslog. Information that can be recorded includes number of times files or directories are accessed, and performance measures such as overall time to perform various vnode operations. This file system can serve as an optimizing or profiling tool for other stackable file systems, by identifying potential bottlenecks.

B.2.8 Regexpfs

A file system that will hide certain files whose names match a regular expression. It could choose to hide these files only from certain users. There are times when you wish to provide access to certain directories but only to a few users. Other times you want to totally hide the existence of some directories or files (exams, proprietary mail, salaries, etc.) from anyone but yourself and the operators performing backups.[9]

More generally, there are times when you want to perform certain operations only on some files, perhaps as few as a single file. Having the ability in a file system to be as granular as one file can be very useful. The main fashion by which this file system operates is when looking up files names, it decides what regular expression matched the file name, and then can classify the file in question as one of several types, each of which can be passed on to be operated upon by a different file system: compressed files can be passed to Gzipfs, encrypted files can get decrypted automatically, and so on.

B.2.9 Unionfs

A file system that presents the union of all the files and directories of several file systems. Special mount options are needed to define the semantics of collision resolution [Pendry95].

[9] This is what's called "security by obscurity."

B.2.10 Hlfs

A file system that uses the user credentials (primarily uid and gid) to create symbolic links different for each user and group, much the way Hlfsd does [Zadok93b]. Hlfs could be used in conjunction with a cryptographic file system to provide user-specific encryption file systems.

B.2.11 Automountfs

A file system that would perform automounter functions much like Amd [Pendry91] does, but in the kernel. It can therefore avoid locking and work much faster. There is only one problem: Amd as it stands knows about the underlying types of file systems that it automounts. If Automountfs will have to know the same, it will violate the symmetry principle of stackable file systems. One solution is to move only part of the automounter code into the kernel, and keep the mount-specific code outside the kernel. This is exactly what Sun had done with Autofs [Callaghan93]: most of the code was moved into the kernel, but the actual mounting is initiated by a user-level daemon called `automountd`. Autofs talks to this daemon using RPCs initiated from the kernel. My file systems would be able to make use of kernel based RPCs to communicated with user-level (or remote) servers.

B.2.12 Applicfs

A file system that would provide per application vnode operations. It is similar to Hlfs described above, with the difference that now, different file system semantics are based on the process ID of the calling context. The information on the current process executing the system call is trivially available in any running kernel.

B.2.13 Namefs

A file system that for every file ever looked up or opened, it keeps the name of that file. This could be useful by other stackable file systems that need to know file names later than when they were originally looked up. This could for example be used in work such as Zadok and Duchamp's [Zadok93a] where the need arose for mapping open vnodes to their pathnames for purposes of simple replication.

B.3 Persistent File Systems

Persistent file systems contain state that should not be lost; therefore it must be written to permanent media. Generally the state would be stored on a local hard-disk, but remote file servers can be used just as easily.

B.3.1 Cachefs

This is very similar to Sun's Cachefs [SunSoft94]. However, Sun's implementation allows for writes through the cache. For simplicity, my initial implementation would pass writing operations directly to the source file system being cached.

B.3.2 Replicfs

A simple replicated (mirroring) file system. This file system will use several file systems assumed to be "identical." Reading operations will be sent to any of the replicas, presumably whichever is the most available at the moment. Writing operations will be performed on all replicas, to ensure consistency. The Statefs file system will be used to store state useful for recovery such as partial write failures, which replicas have the latest versions of the same file, etc.

B.3.3 Expirefs

A file system that will set an expiration date for the files within. This additional information will be recorded using Statefs. A file which expired will be a good candidate for removal. This file system is useful for a multi-user shared temporary space, for USENET news articles that need to get removed automatically after articles expire, and more.

There is one serious problem with such a file system. There is no convenient way to pass expiration date information between user-level processes and the in-kernel file system. Vnode operations such as `vn_getattr` return predetermined information such as uid, gid, file size, last access time, last modification time, last create/mode-change time, etc. The information being passed cannot be changed. Some implementation have left a few empty bytes in this attributes structure, meant for later use. So I could use it for my additional information, but that would not be portable or a vendor supported option for long term use.

The best method for manipulating this information is for Expirefs to provide an additional mount point, besides the one it directly interposes upon. The "shadow" mount point will have a different vnode operations vector (this alone may qualify it to become a different file system) that will provide a file for every file in the file system being interposed. These "dummy" files would have no real storage space associated with them, only a real inode. One of the three time fields (access, modification, creation) of the inode will be used to record the expiration date of the file. That way, programs like `ls` and `find` can continue to function almost normally.

B.3.4 Createfs

This file system will record the *real* creation date of a file, much the same way Expirefs works. While Unix inodes contain a time field called "creation date" that gets initialized at file creation date, this field gets updated each time the file is recreated (via `creat(2)`), update via `touch(1)`, or its mode changed via `chmod(2)`.

There are many times when the real and original creation date of the file is needed and yet current Unix file systems do not keep this information very reliably; there is no way to tell if the creation time stored in the inode is the original one or not. This file system can fix this problem.

B.3.5 Versionfs

A file system that will record version numbers for files each time they are modified. That is, it will record the number of times a file got modified. It may or may not keep backup copies of some older versions of the file.

An alternative way would be to allow the user to set (via similar mechanisms as with Expirefs) the explicit version of the file. The actual version information could be stored in one of the inode fields in the "dummy" file system, and maintained by Statefs.

File versions are very useful. For example, when using replicated file systems, it is often not enough to compare file sizes and dates as a method of ensuring file equivalence. A true file version number, could be a much more efficient and reliable method to tell that, for example, one binary of `emacs` is for version 19.33, and another is for version 19.34. For an expanded discussion on file equivalence in a replicated environment, see [Zadok93a].

A special use for Versionfs would be a file system that is used by multiple software developers to manage source files in a large software project. Such a file system could remove the need to use tools such as RCS or CVS.

Another possible feature of Versionfs might be to change the behavior of `unlink()` such that when a file is removed, a previous version of it is being placed instead. Only when the oldest version of the file is removed, does the file get unlinked from the underlying file system.

B.3.6 Undofs

A file system that will allow a limited form of undoing destructive operations. Unix users often remove files unintentionally.[10] Files that get removed will first be copied over to the backup storage (using Statefs). These files can get expired (perhaps via Expirefs) after a period of disuse. But in the short term, a user realizing the unintentional loss of his/her files could simply copy them from the undo file system back to their original location.

It is important that Undofs will *not* allow non-root users to delete files from the backup location, so that they could not be inadvertently removed.

B.3.7 Aclfs

Although the current vnode interface shown in Appendix section A.4 includes operations on ACLs, these are very rarely used (I know of none). Aclfs is a file system with a simpler form of Access Control Lists. The ACLs will be stored using Statefs. ACLs could for example include information such as *sets* of Unix groups that are allowed to access certain files, sets of users all of which will be treated as owners of the files, and even negation ACLs — users whose membership in certain groups denies them access. It is generally believed that Unix owner and group access permissions are too limiting for multi-user environments, especially software development environments.

B.3.8 Umaskfs

A file system that allows the user to set a per-directory umask. Unix masks are usually set once per the user's environment. Some, like myself, prefer a restrictive umask of 077. But when working in a group on a software project (using say RCS or CVS), it is necessary to set a less restrictive umask of 022 or even 002, allowing all users to read the files created, or users in the group to also write these files. Umaskfs could solve this problem by allowing the user to set a mask for each directory independently.

[10] At least once, we all have intended to type `rm *.o`, but instead typed `rm * .o`.

C Extended Examples Using FiST

In this section I provide three extended examples of file systems designed using FiST, each progressively more complex than the previous. The first is Crossfs, a stateless file system described in Appendix B.1.2. The second is Gzipfs, an in-core file system described in Appendix B.2.5. The third is Replicfs, a persistent file system described in Appendix B.3.2. The keen reader would notice that complicated compilation is not necessary for converting FiST inputs to working C code, only sophisticated, yet straightforward translation.

C.1 Crossfs: A Stateless File System

Crossfs is a trivial file system based on my Null file system (Appendix B.1.1). When a lookup operation crosses into this file system, it performs a simple action such as logging a message on the system console. For all other vnode and vfs operations, it forwards them to the interposed file system. Crossfs keeps no state.

The example of Figure 30 shows the FiST input for this file system. Specifically, in this implementation I wish to log the user ID and the host from where access to the files originated.

```
%{
#ifdef HAVE_AC_CONFIG_H
/* include Autoconf-generated header */
# include "config.h"
#endif
%}

%fstype stateless
%filter syslog(%cur_uid, %from_host) $$ vn_lookup {%cur_uid > 999 && %cur_uid != 2301}

%%
  /* Empty FiST rules section */
%%
  /* No additional code needed */
```

Figure 30: FiST Definition for Crossfs

The code automatically generated for Crossfs will be identical to Nullfs, with the exception of the lookup function. One possible code for the lookup function is shown in Figure 31.

This example shows how FiST "%" directives get translated into local variables (`name`), global variables (`curtime`), or even special functions (`fist_get_from_host()`).

Figure 32 shows the code that would be generated for the NFS version of the same lookup operation.

```
static int
fist_crossfs_lookup( vnode_t *dvp, char *name, vnode_t **vpp,
                     pathname_t *pnp, int flags, vnode_t *rdir, cred_t *cr)
{
  /* check if event should be logged */
  if (u.u_uid > 999 && u.u_uid != 2301)
    kernel_syslog("File %s was accessed at %d by %s@%s.\n",
                  name, curtime, u.u_uid, fist_get_from_host(u));

  /* pass operation to file system, and return status */
  return VOP_LOOKUP(dvp, name, vpp, pnp, flags, rdir, cr);
}
```

Figure 31: Vnode Code Automatically Generated by FiST for Crossfs

```
diropres *
nfsproc2_crossfs_lookup( diropargs *argp,
                         struct svc_req *rqstp;
{
  diropres res;
  uid_t uid;
  gid_t gid;
  char host[MAXHOSTNAMELEN];
  time_t tm;

  /* get credentials */
  if (fist_getcreds(rqstp, &uid, &gid, &host) < 0)
    return(NULL);

  /* get time */
  time(&tm);

  /* check if event should be logged */
  if (uid > 999 && uid != 2301)
    syslog("File %s was accessed at %d by %s@%s.\n",
           argp->name, ctime(&tm), uid, host);

  /* perform generic lookup operation, and return status */
  res = fist_nfs_lookup(argp, rqstp);

  return &res;
}
```

Figure 32: NFS Code Automatically Generated by FiST for Crossfs

C.2 Gzipfs: An In-Core File System

```
%{
#ifdef HAVE_AC_CONFIG_H
/* include Autoconf-generated header */
# include "config.h"
#endif
%}

%fstype incore
%filter gzip $$ %vn_write {%name =~ "\.txt$"}
%filter gunzip $$ %vn_read {%name =~ "\.txt$"}

%%
  /* Empty FiST rules section */
%%
  /* No additional code needed */
```

Figure 33: FiST Definition for Gzipfs

Gzipfs is a compression file system based on my wrapper file system (Appendix B.2.1). Data gets compressed before written to stable media, and decompressed after having been read from such. For this example, I only wish to compress regular files that have a file extension `.txt'`, since ASCII files yield better compression ratios. The example of Figure 33 shows the FiST input for this file system.

The code automatically generated for Gzipfs will be similar to Wrapfs, with the two exceptions of the read and write functions. One possible code for these, for example the `read()` function, is shown in Figure 34.

In this example, the routine decodes the "hidden" vnode pointer, and then passes the read operation to it. After the read had succeeded, we call the FiST filter function `fist_filter_gzip_data()`. This filter is used to decompress data in the *uio*. The filter function would make use of kernel functions that manipulate uio structures such as `uiomove()` to move blocks of bytes between one uio and another. Bytes will be read off of one uio structure, passed through a generic stream decompression function I pulled out of the GNU Zip package, and written to a new uio structure. Then the old uio structure is deallocated and replaced with the new one.

Figure 35 shows the code that would be generated for the NFS version of the same read operation.

```
static int
fist_gzipfs_read(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr)
{
  int error;
  vnode_t *interposed_vp;
  uio_t *new_uiop; /* for the decompressed bytes */

  /* find interposed vnode that is "hidden" inside this vnode */
  interposed_vp = vntofwn(vp)->fwn_vnodep;

  /* pass operation to interposed file system, and return status */
  if ((error = VOP_READ(interposed_vp, uiop, ioflag, cr)) != 0)
    return (error);

  /* Check for triggered events after reading */
  if (regexp_match(fist_get_file_name(vp), "\.txt$"))
    if (fist_filter_gunzip_data(&uiop, sizeof(uio_t), &new_uiop) < 0)
      return EIO; /* I/O error occurred */

  uiop = new_uiop; /* pass up decompressed data */
  return (error);
}
```

Figure 34: Vnode Code Automatically Generated by FiST for Gzipfs

```
int
nfsproc2_gzipfs_read(struct nfsreadargs *in,  struct nfsrdresult *out,
                     struct exportinfo *ex, struct svc_req *sr, cred_t *cr)
{
  int error NFS_OK;

  /* perform simple read */
  error = fist_gzipfs_read(in, out);
  if (error)
    return (error);

  /* check for triggered events after reading */
  if (regexp_match(fist_get_nfs_file_name(in->ra_fhandle), "\.txt$"))
    if (fist_filter_gunzip_nfs_data(in, out) < 0)
      return NFS_ERR;           /* I/O error occurred */

  return (error);
}
```

Figure 35: NFS Code Automatically Generated by FiST for Gzipfs

C.3 Replicfs: A Persistent File System

Replicfs is a persistent file system that replicates files among two copies, as described in Appendix B.3.2. It uses an auxiliary state file system for storing which replica has the most up-to-date copy of each file. For the purpose of this example, I've set the following additional criteria:

- There are only two replicas.
- The state storing file system will record the numeric index number of the file system that has the most up-to-date copy of the file. That number would be 1 for the first replica and 2 for the second replica.
- The key for looking up a file in Statefs' tables is the file ID generated by the vnode operation `vn_fid`. That function generates a unique ID for every file.
- If both replicas are identical, Statefs will not have an entry at all.
- When performing vnode reading operations, call the operation on the "best" replica as recorded in the state. If both replicas are identical, call one of them randomly (thus distributing the operations among both).
- When performing vnode writing operations, call the operation on both replicas in order. If both succeeded, remove the Statefs entry. If only one succeeded, store its number in the state. However, if at least one replica got updated, then do not return an error code; instead, report success.

Of course, these criteria can be changed by the file system's designer to result in different file system semantics. Figure 36 shows the top FiST definitions for Replicfs. Figure 37 shows the FiST rule section for reading operations, and Figure 38 shows the FiST rule section for writing operations.

```
%{
#ifdef HAVE_AC_CONFIG_H
/* include Autoconf-generated header */
# include "config.h"
#endif
%}

%fstype persistent
%interposers 2

%%
```

Figure 36: FiST Definition for Replicfs (top)

Figure 39 shows the code that will be automatically generated by FiST for the reading operation `vn_getattr` (get file attributes).

Figure 40 shows the code that will be automatically generated by FiST for the writing operation `vn_setattr` (set file attributes).

```
/* FiST Rules for read operations*/

$$.%vn_op_read.%variables: {
  int best_copy = 0;
}

$$.%vn_op_read.%in_state: {
  /* find who has the best copy */
  best_copy = %state get, $$.%fid;
};

$$.%vn_op_read.%action: {
  /* perform the operation on the "best" copy */
  if (best_copy == 1) {
    /* first replica is most up-to-date */
    error = $1.%vn_op_this;
  } else if (best_copy == 2) {
    /* second replica is most up-to-date */
    error = $2.%vn_op_this;
  } else {
    /* both replicas are OK. pick one */
    if (fist_random_int() & 0x1 == 0)
      error = $1.%vn_op_this;
    else
      error = $2.%vn_op_this;
  }
}
```

Figure 37: FiST Definition for Replicfs (reading operations)

```
$$.%vn_op_write.%action: {
  retval[1] = $1.%vn_op_this;
  retval[2] = $2.%vn_op_this;
}

$$.%vn_op_write.%error_action: {
  if (retval[1] != 0 && retval[2] != 0) {
    /* both actions failed */
    error = retval[1];
  } else if (retval[1] == 0 && retval[2] != 0) {
    /* replica 2 failed. save "1" in statefs */
    %state add, $$.%vn_fid, 1;
    error = retval[1];
  } else if (retval[1] != 0 && retval[2] == 0) {
    /* replica 1 failed. save "2" in statefs */
    %state add, $$.%vn_fid, 2;
    error = retval[2];
  }
}

$$.%vn_op_write.%out_state: {
  /* both actions succeeded. delete state if any */
  %state del, $$.%vn_fid;
}

%%
/* No additional code needed */
```

Figure 38: FiST Definition for Replicfs (writing operations)

```
static int
fist_wrap_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr)

  int error = EPERM;
  vnode_t *interposed_vp1, *interposed_vp2;
  int best_copy = 0;

  /* lock the interposition chain (default action) */
  fist_lock_interposition_chain(vp);

  /* find the interposed vnodes (default action) */
  interposed_vp1 = vntofwn(vp)->fwn_vnodep1;
  interposed_vp2 = vntofwn(vp)->fwn_vnodep2;

  /* find who has the best copy */
  best_copy = fist_state_get(vp, fist_get_fid(vp));

  /* perform the operation on the "best" copy */
  if (best_copy == 1) {
    /* first replica is most up-to-date */
    error = VOP_GETATTR(interposed_vp1, vap, flags, cr);
  } else if (best_copy == 2) {
    /* second replica is most up-to-date */
    error = VOP_GETATTR(interposed_vp2, vap, flags, cr);
  } else {
    /* both replicas are OK. pick one */
    if (fist_random_int() & 0x1 == 0)
      error = VOP_GETATTR(interposed_vp1, vap, flags, cr);
    else
      error = VOP_GETATTR(interposed_vp2, vap, flags, cr);
  }

  /* unlock the interposition chain (default action) */
  fist_unlock_interposition_chain(vp);

  /* return status code (default action) */
  return (error);
}
```

Figure 39: Vnode Code Automatically Generated by FiST for replicfs (reading operation)

```
static int
fist_wrap_setattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr)
{
  int error = EPERM;
  vnode_t *interposed_vp1, *interposed_vp2;
  int retval[2];

  /* lock the interposition chain (default action) */
  fist_lock_interposition_chain(vp);

  /* find the interposed vnodes (default action) */
  interposed_vp1 = vntofwn(vp)->fwn_vnodep1;
  interposed_vp2 = vntofwn(vp)->fwn_vnodep2;

  /* perform actions on interposed vnodes */
  retval[1] = VOP_SETATTR(interposed_vp1, vap, flags, cr);
  retval[2] = VOP_SETATTR(interposed_vp2, vap, flags, cr);

  /* check if any errors occurred (default action) */
  if (retval[1] != 0 || retval[2] != 0) {
    if (retval[1] != 0 && retval[2] != 0) {
      /* both actions failed */
      error = retval[1];
    } else if (retval[1] == 0 && retval[2] != 0) {
      /* replica 2 failed. save "1" in statefs */
      fist_state_add(vp, fist_get_fid(vp), 1);
      error = retval[1];
    } else if (retval[1] != 0 && retval[2] == 0) {
      /* replica 1 failed. save "2" in statefs */
      fist_state_add(vp, fist_get_fid(vp), 2);
      error = retval[2];
    }

    /* return status code (default action) */
    return (error);
  }

  /* both actions succeeded. delete state if any */
  fist_state_del(vp, fist_get_fid(vp));

  /* unlock the interposition chain (default action) */
  fist_unlock_interposition_chain(vp);

  /* return status code (default action) */
  return (error);
}
```

Figure 40: Vnode Code Automatically Generated by FiST for replicfs (writing operation)

D Appendix: Wrapfs Mount Code

This section includes the actual C code that is used to mount an interposer file system on an interposed one, and is described in Section 3.5.

```
static int
fist_wrap_mount(
        vfs_t *vfsp,            /* pre-made vfs structure to mount */
        vnode_t *vp,            /* existing vnode to mount on */
        struct mounta *uap,     /* user-area mount(2) arguments */
        cred_t *cr              /* user credentials */
)
{
  int error = 0;
#ifdef HAVE_FIST_ARGS
  struct fist_wrap_args args;
  char datalen = uap->datalen;
#endif
  struct fist_wrapinfo *fwip;
  fist_wrapnode_t *fwnp;
  dev_t fist_wrapfs_dev;
  struct vnode *rootvp;
  vnode_t *interposed_vp;       /* interposed vnode */

#ifdef FIST_WRAPDEBUG
  if (vfsp) {
    fist_wrap_print_vfs("fist_wrap_mount", vfsp);
  }
  if (vp) {
    fist_wrap_dprint(fist_wrapdebug, 4,
                     "%s: fist_wrap_vnodeops %x\n",
                     "fist_wrap_mount",
                     (int) &fist_wrap_vnodeops);
    fist_wrap_print_vnode("fist_wrap_mount", vp);
  }
  if (uap) {
    fist_wrap_print_uap("fist_wrap_mount", uap);
  }
#endif

  /*
   * Make sure we're root
   */
  if (!suser(cr)) {
    error = EPERM;
```

```
  goto out;
}

/* Make sure we mount on a directory */
if (vp->v_type != VDIR) {
  error = ENOTDIR;
  goto out;
}

/*
 * check if vnode is already a root of a file system (i.e., there
 * is already a mount on this vnode).
 */
mutex_enter(&vp->v_lock);
if ((uap->flags & MS_REMOUNT) == 0 &&
    (uap->flags & MS_OVERLAY) == 0 &&
    (vp->v_count != 1 || (vp->v_flag & VROOT))) {
  mutex_exit(&vp->v_lock);
  error = EBUSY;
  goto out;
}
mutex_exit(&vp->v_lock);

/*
 * Get arguments: (not needed yet)
 */

/*
 * Get vnode for interposed directory.
 */

/* make sure special dir is a valid absolute pathname string */
if (!uap || !uap->spec || uap->spec[0] != '/') {
  error = EINVAL;
  goto out;
}
error = lookupname(uap->spec, UIO_USERSPACE, FOLLOW,
                   NULLVPP, &interposed_vp);
if (error)
  goto out;
/* Make sure the thing we just looked up is a directory */
if (interposed_vp->v_type != VDIR) {
  VN_RELE(interposed_vp);
  error = ENOTDIR;
```

```
    goto out;
  }
#ifdef FIST_WRAPDEBUG
  if (interposed_vp) {
    fist_wrap_print_vnode("fist_wrap_mount", vp);
  }
#endif

  /*
   * Now we can increment the count of module instances.
   * meaning that from now, the mounting cannot fail.
   */
  ++module_keepcnt;


  /*****************************************************************************
   * FIST_WRAPINFO:
   * The private information stored by the vfs for fist_wrapfs.
   */
  /* this implicitly allocates one vnode to be used for root vnode */
  /* XXX: enter this vnode in dnlc? */
  fwip = (struct fist_wrapinfo *)
    kmem_alloc(sizeof(struct fist_wrapinfo), KM_SLEEP);
  /* store the vfs of the stacked file system (pushed onto "stack") */
  fwip->fwi_mountvfs = vp->v_vfsp;
  /* initialize number of interposed vnodes */
  fwip->fwi_num_vnodes = 0;
  /* fwip->fwi_rootvnode: is setup in the "root vnode" section below */


  /*****************************************************************************
   * FIST_WRAPNODE:
   * The private information stored by interposing vnodes.
   * The interposing vnode here is the new root vnode of fist_wrapfs.  It
   * interposes upon the uap->spec vnode we are mounting on (the directory,
   * or partition interposed upon).
   */
  fwnp = (fist_wrapnode_t *)
    kmem_alloc(sizeof(fist_wrapnode_t), KM_SLEEP);
  fwnp->fwn_vnodep = interposed_vp;


  /*****************************************************************************
   * VFS FOR THE FIST_WRAP FILE SYSTEM:
```

```
 */
vfsp->vfs_bsize = 1024;
vfsp->vfs_fstype = fist_wrapfs_fstype;
/* Assign a unique device id to the mount */
mutex_enter(&fist_wrapfs_minor_lock);
do {
  fist_wrapfs_minor = (fist_wrapfs_minor + 1) & MAXMIN;
  fist_wrapfs_dev = makedevice(fist_wrapfs_major, fist_wrapfs_minor);
} while (vfs_devsearch(fist_wrapfs_dev));
mutex_exit(&fist_wrapfs_minor_lock);
/* set the rest of the fields */
vfsp->vfs_dev = fist_wrapfs_dev;
vfsp->vfs_fsid.val[0] = fist_wrapfs_dev;
vfsp->vfs_fsid.val[1] = fist_wrapfs_fstype;
vfsp->vfs_bcount = 0;
/* store private fist_wrap info in the pre-made vfs */
vfsp->vfs_data = (caddr_t) fwip;
/* fill in the vnode we are mounted on, in the vfs */
vfsp->vfs_vnodecovered = vp;


/*****************************************************************************
 * ROOT VNODE OF FIST_WRAPFS:
 */
rootvp = &(fwip->fwi_rootvnode);
VN_INIT(rootvp, vfsp, VDIR, (dev_t) NULL);
/* this is a root vnode of this file system */
rootvp->v_flag |= VROOT;
/* vnode operations of this root vnode are the fist_wrap */
rootvp->v_op = &fist_wrap_vnodeops;
/* this one is NOT a mount point at this stage */
rootvp->v_vfsmountedhere = NULL;

/*
 * This v_data stores the interposed vnode in for now, but in the future
 * it could hold more information which is specific to a single vnode
 * within a file system.  For example, in fist_gzipfs, we could store
 * information about the file: type of compression (gzip, pack, zip, lzh,
 * compress, etc), whether the file should not be compressed (maybe it is
 * stored already in a compact format such as GIF files), etc.
 */
rootvp->v_data = (caddr_t) fwnp;
/* NULLify the rest, just in case */
rootvp->v_filocks = NULL;
```

```
  /*   rootvp->v_cv = NULL; */ /* don't do this one for now */


  /*************************************************************************
   * VNODE MOUNTED UPON:
   */
  /* this vnode to mount on is a mount point for fist_wrap */
  vp->v_vfsmountedhere = vfsp;


#ifdef FIST_WRAPDEBUG
  /* print values after we change them */
  if (vfsp) {
    fist_wrap_print_vfs("fist_wrap_mount2", vfsp);
  }
  if (vp) {
    fist_wrap_print_vnode("fist_wrap_mount2", vp);
  }
  fist_wrap_print_vnode("fist_wrap_mount2rvn", &(fwip->fwi_rootvnode));
#endif

out:
  /*
   * Cleanup our mess
   */
#ifdef FIST_WRAPDEBUG
  fist_wrap_dprint(fist_wrapdebug, 4, "fist_wrap_mount: EXIT\n");
#endif
  return (error);
}
```

E Appendix: Portability Using Autoconf

Source code portability is not easy to achieve across Unix platforms. Many variants use completely different and incompatible interfaces. Especially difficult to port is "system" code — that is, code that performs low level operations such as network messaging, system calls, file serving, access to kernel data structures, and of course, kernel resident code itself.

E.1 Portability Solutions

Several solutions to the problem of portability were used over the years. The simplest was to include header files with each package that abstract away the differences between platforms using a plethora of multi-nested `#define` and `#ifdef` statements. It made code very hard to read. Other alternatives asked the user to run an interactive configuration script that prompted the user to answer questions such as "Is this machine big-endian?" and "Are you POSIX Compliant?" These configuration scripts tended to become very long, verbose, and tedious for users to go through. Worse of all, they did not guarantee that the user would really answer the questions correctly. To answer some of them correctly one had to be a Unix expert to begin with. More sophisticated solutions used the X11 Imake utility which abstracted the differences using preprocessing (via `cpp`) of several pre-written template files. Imake's usefulness never extended beyond that of the X11 domain of applications [Haemer94].

All of these solutions suffered from one major problem — they were static. That is, the portability offered was only as good as what the programmers of the package included. They could not be easily changed to accommodate new operating systems or even new minor revisions of existing operating systems. In addition, they could never account for partially installed or *misinstalled* systems. For example, operating systems such as Solaris and IRIX require the installation of special software packages in order to use Motif or NFS, respectively. System administrators could choose to install these packages or not. It is even possible (and unfortunately quite common), for systems to claim to have a particular feature but not to implement it correctly. Finally, Unix systems are as good as the administrators who maintain them. Often, complex installations tend to have poor configurations. A good solution to portability must be able to handle all of these cases.

The Free Software Foundation (FSF) solved these problems using a dynamic, automatic configuration system called Autoconf [MacKenzie95], which I plan to use with FiST. Autoconf is a large collection of highly portable M4 macros and Bourne shell scripts that perform on-the-fly feature tests to determine differences among systems.

For example, in order to find out if one has the proper Motif libraries to link X11 applications with, Autoconf provides a simple test that can be used as follows: `AC_CHECK_LIB(Xm)`. The test in turn is implement as a small shell script that writes a test C program on the fly, and tries to compile and link it. If it succeeds, it knows for certain that the Motif library libXm is available. If the test is successful, then Autoconf modifies the auto-generated Makefile and adds to it the line `LIBS += -lXm`. The Makefile generated is guaranteed to link with the Motif library if and only if it exists.

Another example is the Autoconf macro `AC_FUNC_ALLOCA`. It runs tests that check for the existence of the `alloca(3)` library call. This particular library call is known to have many broken implementations on various systems. Autoconf therefore performs additional tests to validate the correct behavior of the call! If successful, Autoconf will add the line `#define HAVE_ALLOCA_H` to the autogenerated header file it creates, `"config.h"`. An application can include this locally created header file and use the

definitions within to ensure that the proper headers and their associated definitions get included, and nothing more.

Autoconf's standard M4 tests include easy facilities to extend them: you can supply actions to be performed if the test failed or succeeded, you can include existing tests in other tests you write, you can cache previous results and reuse them, and so on. The basic set of Autoconf tests have been used by large and very complex packages such as the GNU HURD, *gcc*, *emacs*, *gdb*, LaTeX, Tcl/Tk, and many more. Autoconf can make it easy to port applications to over one hundred Unix variants known, and by its nature automatically handles new ones as they spring into existence.

E.2 An Example Using Autoconf

Here is an example of how I intend to use Autoconf within FiST. The name of the VFS structure on most operating systems such as SunOS and Solaris is `struct vfs` and is defined `<sys/vfs.h>`. But on other systems such as FreeBSD, the name of the same structure is `struct mount` and is defined in `<sys/mount.h>`. Existing tests within Autoconf can find out if a C structure named `vfs` is defined in any of the system header files. If not found, the failure action code for looking up `struct vfs` would invoke the same test, but on a different name: it would look for `struct mount`. Once found, Autoconf will create a typedef which will be one of these two: `typedef struct vfs vfs_t;` or `typedef struct mount vfs_t;`. In addition, Autoconf will define *one* of `#define HAVE_SYS_VFS_H` or `#define HAVE_SYS_MOUNT_H` in the `config.h` file it creates. I would then write code that includes the correct header file and uses the typedef whenever I need to refer to the VFS structure. Figure 41 shows how I will write such VFS code.

In a similar manner I will write Autoconf tests that find and generalize more minute differences such as the different names used for fields within key C structures, whether an operating system has loadable kernel modules or not, what macros are used to dereference VFS and vnode pointers, and so on.

```
#ifdef HAVE_AC_CONFIG_H
/* include Autoconf-generated header */
# include "config.h"
#endif

#ifdef HAVE_SYS_VNODE_H
# include <sys/vnode.h>
#endif

#ifdef HAVE_SYS_MOUNT_H
# include <sys/mount.h>
#endif

int print_vfs(vfs_t *vfsp)
{
  /* code to print values within the vfs structure */
}
```

Figure 41: VFS Sample Code Using Autoconf

Autoconf can perform syntactic checks and limited tests for the correct use of certain symbols based on syntactic features. Autoconf, however, cannot solve purely semantic problems. Without additional help, it cannot discover the *meaning* of, say, two symbols with the same name across different operating systems that are used differently. Those cases unfortunately have to be specially handled. Nevertheless, Autoconf is a tool that will be able to figure out over 95% of the differences among operating systems automatically.

References

[Abrosimov92] V. Abrosimov, F. Armand, and M. I. Ortega. A Distributed Consistency Server for the CHORUS System. *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)* (Newport Beach, CA), pages 129–48. USENIX, 26-27 March 1992.

[Accetta86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *USENIX Conference Proceedings* (Atlanta, GA), pages 93–112. USENIX, Summer 1986.

[Anderson92] D. P. Anderson, Y. Osawa, and R. Govindan. The Continuous Media File System (Real-Time Disk Storage and Retrieval of Digital Audio/Video Data). *USENIX Conference Proceedings* (San Antonio, TX), pages 157–64. USENIX, Summer 1992.

[Bershad88] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. *USENIX Conference Proceedings* (Dallas, TX), pages 267–75. USENIX, Winter 1988.

[Bishop88] M. A. Bishop. Auditing Files on a Network of UNIX Machines. *UNIX Security Workshop* (Portland, Oregon), pages 51–2. USENIX, 29-30 August 1988.

[Blaze93] M. Blaze. A Cryptographic File System for Unix. *Proceedings of the first ACM Conference on Computer and Communications Security* (Fairfax, VA). ACM, November, 1993.

[Boneh96] D. Boneh and R. J. Lipton. A Revocable Backup System. *The Sixth USENIX UNIX Security Symposium Proceedings* (San Jose, California), pages 91–6. USENIX, 22-25 July 1996.

[Bosch96] P. Bosch and S. J. Mullender. Cut-and-paste file-systems: integrating simulators and file-systems. *USENIX* (San Diego, CA), January 1996.

[Breitstein97] S. R. Breitstein. Inferno Namespaces. Online White-Paper. Lucent Technologies, 11 March 1997. Available via the WWW in `http://www.inferno.lucent.com/namespace.html`.

[Callaghan89] B. Callaghan and T. Lyon. The Automounter. *USENIX Conference Proceedings* (San Diego, CA), pages 43–51. USENIX, Winter 1989.

[Callaghan93] B. Callaghan and S. Singh. The Autofs Automounter. *USENIX Conference Proceedings* (Cincinnati, OH), pages 59–68. USENIX, Summer 1993.

[Cate92] V. Cate. Alex – a global filesystem. *Proceedings of the File Systems Workshop* (Ann Arbor, MI), pages 1–11. Usenix Association, 21-22 May 1992.

[CORBA91] D. E. Corp., H.-P. Company, H. Corp., N. Corp., O. D. Inc., and S. Inc. *The Common Object Request Broker: Architecture and Specification*, OMG document number 91.12.1, Rev. 1.1. Object Management Group, Draft 10 December 1991.

[Elmasri94] R. Elmasri and S. B. Navathe. Dynamic Multilevel Indexes Using B-Trees and B+-trees. In *Fundamentals of Database Systems*, pages 116–28. Benjamin Cummings, 1994.

[Fall94] K. Fall and J. Pasquale. Improving Continuous-Media Playback Performance with In-Kernel Data Paths. *IEEE Conference on Multimedia Computing and Systems*, pages 100–9, May 1994.

[Fitzhardinge94] J. Fitzhardinge. Userfs – user process filesystem. Unpublished software package documentation. Softway Pty, Ltd., August, 1994. Available via ftp from `tsx-11.mit.edu` in `/pub/linux/ALPHA/userfs`.

[Forin94] A. Forin and G. Malan. An MS-DOS Filesystem for UNIX. *USENIX Conference Proceedings* (San Francisco, CA), pages 337–54. USENIX, Winter 1994.

[Gingell87a] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared Libraries in SunOS. *USENIX Conference Proceedings* (Phoenix, AZ), pages 131–45. USENIX, Summer 1987.

[Gingell87b] R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual Memory Architecture in SunOS. *USENIX Conference Proceedings* (Phoenix, AZ), pages 81–94. USENIX, Summer 1987.

[Glover93] F. Glover. A Specification of Trusted NFS (TNFS) Protocol Extensions. Technical report. Internet Engineering Task Force, 1 March 1993.

[Golub90] D. Golub. ddb(4). Mach Operating System Reference Manual, Section 4. Carnegie Mellon University, 8 August 1990.

[Gutmann96] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. *The Sixth USENIX UNIX Security Symposium Proceedings* (San Jose, California), pages 77–89. USENIX, 22-25 July 1996.

[Guy90] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[Haemer94] J. S. Haemer. Imake Rhymes with Mistake. *;login:*, **19**(1):32–3. USENIX, January-February 1994.

[Haynes92] R. A. Haynes and S. M. Kelly. Software Security for a Network Storage Service. *UNIX Security III Symposium Proceedings* (Baltimore, Maryland), pages 253–65. USENIX, 14-16 September 1992.

[Heidemann91] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Technical report CSD-910007. University of California, Los Angeles, March 1991.

[Heidemann94] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *Transactions on Computing Systems*, **12**(1):58–89. (New York, New York), ACM, February, 1994.

[Kao89] P.-H. Kao, B. Gates, B. Thompson, and D. McCluskey. Support of the ISO-9660/HSG CD-ROM File System in HP-UX. *USENIX Conference Proceedings* (Baltimore, MD), pages 189–202. USENIX, Summer 1989.

[Kardel90] F. Kardel. Frozen Files. *UNIX Security II Workshop Proceedings* (Portland, Oregon), pages 83–6. USENIX, 27-28 August 1990.

[Kernighan96] B. W. Kernighan. A Descent into Limbo. Online White-Paper. Lucent Technologies, 12 July 1996. Available via the WWW in `http://www.inferno.lucent.com/inferno/limbotut.html`.

[Khalidi93] Y. A. Khalidi and M. N. Nelson. Extensible File Systems in Spring. *Proceedings of the 14th Symposium on Operating Systems Principles* (Asheville, North Carolina). ACM, December, 1993.

[Kistler91] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar Conference Center, Pacific Grove, U.S.), volume 25, number 5, pages 213–25. ACM Press, 1991.

[Kistler93] J. J. Kistler. Disconnected Operation in a Distributed File System (Report CMU-CS-93-156). Technical report. Carnegie Mellon University, Pittsburgh, U.S., 1993.

[Kleiman86] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *USENIX Conference Proceedings* (Atlanta, GA), pages 238–47. USENIX, Summer 1986.

[Kramer88] S. M. Kramer. On Incorporating Access Control Lists into the UNIX Operating System. *UNIX Security Workshop* (Portland, Oregon), pages 38–48. USENIX, 29-30 August 1988.

[Kuenning94] G. Kuenning, G. J. Popek, and P. Reiher. An Analysis of Trace Data for Predictive File Caching in Mobile Computing. *USENIX Summer 1994 Conference (To Appear)* (Boston, U.S.), 1994.

[Lord96] T. Lord. Subject: SNFS – Scheme-extensible NFS. Unpublished USENET article. emf.net, 30 October 1996. Available via ftp in ftp://emf.net:users/lord/systas-1.1.tar.gz.

[LoVerso91] S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg. The OSF/1 UNIX Filesystem (UFS). *USENIX Conference Proceedings* (Dallas, TX), pages 207–18. USENIX, 21-25 January 1991.

[Lucent97] Lucent. Inferno: la Commedia Interattiva. Online White-Paper. Lucent Technologies, 13 March 1997. Available via the WWW in `http://inferno.lucent.com/inferno/infernosum.html`.

[MacKenzie95] D. MacKenzie. Autoconf: Creating Automatic Configuration Scripts. User Manual, Edition 2.7, for Autoconf version 2.7. Free Software Foundation, November 1995.

[Marsh94] B. Marsh, F. Douglis, and P. Krishnan. Flash Memory File Caching for Mobile Computers. Technical report. Matsushita Information Technology Laboratory, U.S., 1994.

[McKusick84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[Mercer94] C. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–9, May 1994.

[Minnich93] R. G. Minnich. The AutoCacher: A File Cache Which Operates at the NFS Level. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 77–83. USENIX, Winter 1993.

[Mitchel94] J. G. Mitchel, J. J. Giobbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. *CompCon Conference Proceedings* (San Francisco, California). CompCon, 1994.

[Moran93] J. Moran and B. Lyon. The Restore-o-Mounter: The File Motel Revisited. *USENIX Conference Proceedings* (Cincinnati, OH), pages 45–58. USENIX, Summer 1993.

[Mummert95] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. *Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Association for Computing Machinery SIGOPS, 3–6 December 1995.

[Pasquale94] J. Pasquale, E. Anderson, and K. Muller. Container Shipping: Operating System Support for I/O-Intensive Applications. *IEEE Computer*, pages 84–93, March 1994.

[Pawlowski94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. *USENIX Conference Proceedings* (Boston, Massachusetts), pages 137–52. USENIX, 6-10 June 1994.

[Pendry91] J.-S. Pendry and N. Williams. Amd – The 4.4 BSD Automounter. User Manual, edition 5.3 alpha. Imperial College of Science, Technology, and Medicine, London, England, March 1991.

[Pendry95] J.-S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. *Conference Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 25–33. Usenix Association, 16–20 January 1995.

[Pike90] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *Proceedings of Summer UKUUG Conference*, pages 1–9, July 1990.

[Pike91] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9, a distributed system. *Proceedings of Spring EurOpen Conference*, pages 43–50, May 1991.

[PLC96] PLC. StackFS: The Stackable File System Architecture. Online White-Paper. Programmed Logic Corporation, 1996. Available via the WWW in `http://www.plc.com/st-wp.html`.

[Presotto93] D. Presotto and P. Winterbottom. The Organization of Networks in Plan 9. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 271–80. USENIX, Winter 1993.

[Ramakrishnan93] K. Ramakrishnan, L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glasner, and W. Duso. Operating System Support for a Video-on-Demand File Service. *Proceedings of 4th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video* (Lancaster, UK), November 1993.

[Rees86] J. Rees, P. H. Levine, N. Mishkin, and P. J. Leach. An Extensible I/O System. *USENIX Conference Proceedings* (Atlanta, GA), pages 114–25. USENIX, Summer 1986.

[Ritchie84] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, **63**(8):1897–910, October 1984.

[Rodriguez86] R. Rodriguez, M. Koehler, and R. Hyde. The generic file system. *1986 Summer USENIX Technical Conference* (Atlanta, GA, June 1986), pages 260–9. USENIX, June 1986.

[Rosenblum91] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 1–15. Association for Computing Machinery SIGOPS, 13 October 1991.

[Rosenthal90] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conference Proceedings* (Anaheim, CA), pages 107–18. USENIX, Summer 1990.

[Rosenthal92] D. S. H. Rosenthal. Requirements for a "Stacking" Vnode/VFS Interface. Unix International document SD-01-02-N014. UNIX International, 1992.

[Sandberg85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Association Summer Conference Proceedings of 1985* (11-14 June 1985, Portland, OR), pages 119–30. USENIX Association, El Cerrito, CA, 1985.

[Satyanarayanan90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, **39**:447–59, 1990.

[Skinner93] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. *USENIX Conference Proceedings* (Cincinnati, OH), pages 161–74. USENIX, Summer 1993.

[SMCC90] SMCC. syslogd(8). SunOS 4.1 Reference Manual, Section 8. Sun Microsystems, Incorporated, January, 1990.

[SMCC91] SMCC. config(8). SunOS 4.1 Reference Manual, Section 8. Sun Microsystems, Incorporated, 10 April 1991.

[SMCC93a] SMCC. modload(1M). SunOS 5.5 Reference Manual, Section 1M. Sun Microsystems, Incorporated, 1 December 1993.

[SMCC93b] SMCC. Overview of Online: DiskSuite. In *Online DiskSuite 2.0 Administration Guide*, pages 31–6. SunSoft, May, 1993.

[SMCC94a] SMCC. kadb(1M). SunOS 5.4 Reference Manual, Section 1M. Sun Microsystems, Incorporated, 2 June 1994.

[SMCC94b] SMCC. swap(1M). SunOS 5.5 Reference Manual, Section 1M. Sun Microsystems, Incorporated, 2 March 1994.

[SMCC95] SMCC. crash(1M). SunOS 5.5 Reference Manual, Section 1M. Sun Microsystems, Incorporated, 25 January 1995.

[Stallman94] R. M. Stallman and R. H. Pesch. The GNU Source-Level Debugger. User Manual, Edition 4.12, for GDB version 4.13. Free Software Foundation, January 1994.

[Steiner88] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. *USENIX Conference Proceedings* (Dallas, TX), pages 191–202. USENIX, Winter 1988.

[Stewart93] J. N. Stewart. AMD – The Berkeley Automounter, Part 1. *;login:*, **18**(3):19. USENIX, May/June 1993.

[Stone87] D. L. Stone and J. R. Nestor. IDL: background and status. *SIGPLAN Notices*, **22**(11):5–9, November 1987.

[Stonebraker81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, **24**(7):412–18, July 1981.

[Stonebraker86] M. Stonebraker and A. Kumar. Operating system support for data management. *Database Engineering*, **9**(3):43–51, September 1986.

[Sun89] Sun Microsystems, Incorporated. *NFS: Network File System protocol specification*, Technical report RFC–1094, March 1989.

[SunSoft94] SunSoft. Cache File System (CacheFS). A White-Paper. Sun Microsystems, Incorporated, February 1994.

[Tait91] C. Tait and D. Duchamp. Service Interface and Replica Consistency Algorithm for Mobile File System Clients. *1st International Conference on Parallel and Distributed Information Systems*, 1991.

[Tait92] C. D. Tait and D. Duchamp. An Efficient Variable-Consistency Replicated File Service. Technical report. Department of Computer Science, Columbia University, New York, U.S., 1992.

[Takahashi95] T. Takahashi, A. Shimbo, and M. Murota. File-Based Network Collaboration System. *The Fifth USENIX UNIX Security Symposium Proceedings* (Salt Lake City, Utah), pages 95–104. USENIX, 5-7 June 1995.

[Tanenbaum92] A. S. Tanenbaum. The MS-DOS File System. In *Modern Operating Systems*, pages 340–2. Prentice Hall, 1992.

[Warren87] W. B. Warren, J. Kickenson, and R. Snodgrass. A tutorial introduction to using IDL. *SIGPLAN Notices*, **22**(11):18–34, November 1987.

[Zadok93a] E. Zadok and D. Duchamp. Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing. *USENIX Conference Proceedings* (Cincinnati, OH), pages 69–85. USENIX, Summer 1993.

[Zadok93b] E. Zadok and A. Dupuy. HLFSD: Delivering Email to Your $HOME. *Systems Administration (LISA VII) Conference* (Monterey, CA), pages 243–54. USENIX, 1-5 November 1993.