FiST: A Language for Stackable File Systems

Erez Zadok and Jason Nieh
Computer Science Department, Columbia University
{ezk,nieh}@cs.columbia.edu

CUCS-034-1999

Abstract

Stackable file systems promise to ease the development of file systems[6, 16, 19]. Operating system vendors, however, resist making extensive changes to support stacking, because of the impact on performance and stability. Existing file system interfaces differ from system to system and they support extensibility poorly. Consequently, extending file system functionality across platforms is difficult.

We propose a new language, *FiST*, to describe stackable file systems. FiST uses operations common to file system interfaces. From a single description, FiST's compiler produces file system modules for multiple platforms. The generated code handles many kernel details, freeing developers to concentrate on the main issues of their file systems.

This paper describes the design, implementation, and evaluation of FiST. We extended file system functionality in a portable way without changing existing kernels. We built several file systems using FiST on Solaris, FreeBSD, and Linux. Our experiences with these examples shows the following benefits of FiST: average code size over other stackable file systems is reduced ten times; average development time is reduced seven times; performance overhead of stacking is 1–2%.

1 Introduction

Writing new file systems or extending existing ones is difficult. *Stackable file systems*[15] promise to speed the process of file system development by providing an extensible file system interface. This extensibility allows new features to be added incrementally over other stackable file systems.

A number of approaches have been taken toward the design of stackable file systems: new stackable file system interfaces, null layer file systems, source code templates, and stackable templates. Several new *stackable file system interfaces* have been proposed and a few have been implemented[6, 16, 19]. Unfortunately, these proposals required many changes to existing operating systems and file systems. This added overhead to the performance of native file systems, even if these stacking interfaces were not in use. As a result, almost none of the major operating system vendors today offer a stackable file system interface.

Null layer file systems such as Solaris's Lofs[20] or BSD's Nullfs[14] implement some of the stacking principles, and thus are suitable as templates from which other stackable file systems could be written. However, null layer file systems such as Lofs and Nullfs lack crucial infrastructure necessary for practical stacking. They do not provide developers with an easy way to modify file data, change file names, inspect file attributes, add new functionality, etc.

Alternatively, existing source code for any file system (FFS, NFS, etc.) can be used as a starting template for file system development. The code base for existing file systems, however, is usually large and complicated. Again, detailed knowledge of kernel and file system internals is required to write stackable file systems using any other file system as a template.

In earlier work to improve the process of writing stackable file systems, we introduced a stackable template system called Wrapfs[25]. Unlike other stackable file system interfaces, Wrapfs adds missing stacking functionality without changing existing kernels or file systems. This improves the stability of the rest of the system, thus alleviating some of concerns operating system vendors have. Unlike null layer file systems, Wrapfs includes code to manipulate file names and file data—the two most common changes desired by developers. However, like other stackable file systems, writing file systems using Wrapfs still has a number of drawbacks:

- File system developers using Wrapfs are required to modify Wrapfs templates by hand, increasing the chances for human errors.
- Making simple changes to file names or file data is relatively easy with Wrapfs, because Wrapfs centralizes such changes into a few places. Making other changes, even small ones, is sometimes tedious: with thousands of lines of C code per Wrapfs template (Section 5.1), developers have to read through many lines of operating system specific code.
- Wrapfs does not alleviate many portability concerns. Developers still have to find, for each platform, how to allocate and free kernel memory, how to access or modify user credentials, how to change file attributes, etc. Code written in one Wrapfs template for one platform has to be ported to all other operating system templates to run on other platforms. The more Wrapfs templates there are (currently four), the more porting work is required.
- Because Wrapfs provides a general stackable file system infrastructure, it provides functionality that may not be used by all file systems. If developers do not need to make changes to file names or file data, they have to manually and carefully remove all of that code, and replace it with appropriate code. If that unused code is not removed, many unnecessary identical copies of data pages and file names are created, consuming memory and affecting performance unnecessarily. Moreover, kernel modules containing unused code consume precious kernel memory (which is often physical memory).

Programmers who want to try a new file system feature, even a small one, must still inspect and modify large amounts of existing code. This effort must be repeated for each port to another platform. Differences in file system interfaces and operating systems make portability of file systems very difficult. Even if one uses the platform specific source code for the same file system (e.g., NFS) as a template, one still finds that actual implementations across platforms differ significantly. As a result, application developers, who are neither systems programmers nor file system developers, lack the tools necessary to experiment with new file system ideas that may improve their user applications.

1.1 Why a Language?

To ease the problems of developing and porting stackable file systems, we propose a high-level language to describe such file systems. Four benefits to using a language are:

Simplicity: A file system language can provide higher-level primitives and first-class functions specific to file systems that simplify their development and reduce the amount of code that developers need to write. This reduces developers' need to know many kernel internals, allowing even non-experts to develop file systems.

Portability: A language can describe file systems using an interface abstraction that is common to operating systems. The language compiler can bridge the gaps between its interface and the other platforms' interfaces. From a single description of a file system, we could generate file system code for different platforms. This improves portability considerably.

Specialization: A language allows developers to customize the file system their needs. Instead of having one large and complex file system with many features that can be configured and turned on or off, the compiler for a file system language can produce special-purpose file systems. This improves performance and memory footprint because specialized file systems include only necessary code.

Optimization: A language compiler can perform global optimizations to improve the performance of file systems that would be tedious and difficult to do by hand. For example, a language compiler can collapse functionality from multiple stacking layers into a single layer, removing the multiplicative stacking overhead costs of each layer.

This paper describes the design and implementation of the *FiST*, a *File System Translator* language for stackable file systems. FiST lets developers describe stackable file systems at a high level, using operations common to file system interfaces and system calls. With FiST, developers need only describe the core functionality of their file systems. The FiST language code generator, *fistgen*, generates file system modules for several platforms using a single description. We currently support Solaris 2.6, FreeBSD 3.3, and Linux 2.3.

To assist fistgen with generating stackable file systems, we created a minimal stackable file system template called Basefs. Basefs adds missing stacking functionality and relieves fistgen from dealing with many platform-dependent aspects of file systems. Basefs does not require any changes to the kernel or existing file systems. Its main function is to handle many kernel internals relating to stacking: how and when to lock file system objects, how to allocate and free objects, accurately maintaining reference counts, and more. Basefs provides simple hooks for fistgen to insert code that performs common tasks desired by file system developers, such as modifying file data or inspecting file names. That way, fistgen can produce file system code for any platform we port Basefs to. The hooks also allow fistgen to include only necessary code, improving performance and reducing kernel memory usage.

We built several example file systems using FiST. Our experiences with these examples shows the following benefits of FiST compared with other stackable file systems: average code size is reduced ten times; development time is reduced seven times; performance overhead of stacking is less than 2%, and unlike other stacking systems, there is no performance overhead for native file systems.

Our focus in this paper is to demonstrate how FiST simplifies the development of file systems, provides write-once run-anywhere portability across UNIX systems, and reduces stacking overhead through file system specialization. The rest of this paper is organized as follows. Section 2 details the design of FiST, and describes the FiST language, fistgen, and Basefs. Section 3 discusses key implementation and portability details. Section 4 describes four example file systems written using FiST. Section 5 evaluates the ease of development, the portability, and the performance of our file systems. Section 6 surveys related work. Finally, Section 7 concludes and explores future directions.

2 Design

We have five main design goals for the FiST system:

1. The FiST language should allow developers to express easily common file system operations. Detailed understanding of file system internals should not be necessary.
2. FiST code should look familiar to system developers. In addition, FiST's syntax should incorporate the best common file system features from various operating systems and file system interfaces.
3. FiST should take care of as many kernel internals and portability issues as possible.
4. FiST should not limit what file system developers can implement and should provide as much functionality as traditional system programming languages such as C. Furthermore, code produced by FiST should be readable and easily extensible to enable developers to have full control of the produced file system.

5. FiST performance overhead should be small and at least comparable to the best of other stackable file systems.

The overall structure of the FiST system is shown in Figure 1. The figure illustrates how the three parts of FiST work

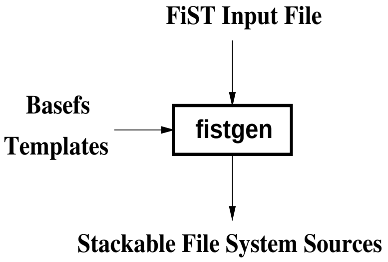

Figure 1: FiST Operational Diagram. Fistgen reads a FiST input file, and with the Basefs templates, produces sources for a new file system.

together: the FiST language, fistgen, and Basefs. File system developers write FiST input files to implement file systems using the FiST language. *Fistgen*, the FiST language code parser and file system code generator, reads FiST input files that describe the new file system's functionality. Fistgen uses additional input files, the Basefs templates. These templates contain the stacking support code for each operating system and hooks to insert developer code. Fistgen combines the functionality described in the FiST input file, with the Basefs templates, and produces new sources as output. These C sources implement the functionality of the new file system. Developers can, for example, write simple FiST code to manipulate file data and file names. Fistgen, in turn, translates that FiST code into C code and inserts it at the right place in the templates, along with any additional support code that may be required. Developers can also turn on or off certain file system features, and fistgen will conditionally include code that implements those features.

Figure 2 shows the hierarchy for different file system abstractions. At the lowest level reside file systems native to the operating system, such as disk based and network based file systems. They are at the lowest level because they interact directly with device drivers. Above native file systems there are stackable file systems such as the examples in Section 4, as well as Basefs and Wrapfs themselves. These file systems provide a higher abstraction that native file systems because stackable file systems interact only with other file systems through a well defined file system interface. At the highest level, we define the FiST language. FiST abstracts the different file system (vnode) interfaces and different operating systems into a single common description language. We found that while vnode *vnode interface*[9] interfaces differ from system to system, they share many similar features. Our experience shows that similar file system concepts exist in other non-Unix systems, and our stacking work can be generalized to include them. Therefore, we designed the FiST language to be as general as possible.

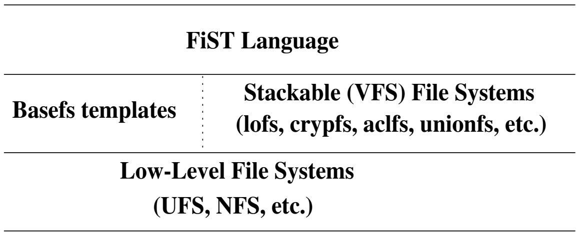

Figure 2: FiST Structural Diagram. Stackable file systems, including Basefs, are at the VFS level, and are above low-level file systems. FiST descriptions provide a higher abstraction than that provided by the VFS.

2.1 The FiST Language

The FiST language is a high-level language that uses file system features common to several operating systems. It provides file system specific language constructs for simplifying file system development. In addition, FiST language constructs can be used in conjunction with additional C code to provide the full flexibility of a system programming language familiar to file system developers. The ability to integrate C and FiST code is reflected in the general structure of FiST input files. Figure 3 shows the four main section of a FiST input file. The FiST grammar was modeled after YACC[7]

1	`%{`
	C Declarations
	`%}`

2	*FiST Declarations* `%%`
3	*FiST Rules* `%%`
4	*Additional C Code*

Figure 3: FiST Grammar Outline

input files, because YACC is familiar to programmers and the purpose for each of its four sections matches with four different subdivisions of desired file system code: raw included header declarations, declarations which affect the produced code globally, actions to perform when matching vnode operations, and additional code.

C Declarations are used to include additional C headers, define macros or typedefs, list forward functions, etc. to be used throughout the rest of the code.

FiST Declarations define global file system properties that affect the overall semantics of the produced code. These properties are useful because they allow developers to make common global changes in a simple manner. In this section, for example, we declare if the file system will be read-only or not. FiST Declarations can also define special data structures used by the rest of the code for this file system.

FiST Rules define per-rule actions. A FiST rule is a piece of code that executes for a selected set of vnode operations, for one operation, or even a portion of a vnode operation. Rules allow developers to control the behavior of one or more file system functions in a portable manner. The FiST rules section is the primary section, where most of the actions for the produced code are written. In this section, for example, we can choose to change the behavior of `unlink` to rename the file to be deleted, so it might be restored later.

Additional C Code includes additional C functions that might be necessary or referenced by code in the rules section. We separated this section from the rules section for code modularity: FiST rules are actions to take for a given vnode function, while the additional C code may contain arbitrary code that could be called from anywhere.

The C Declarations and Additional C Code sections are simple. Next we go into the details of the two more important sections: FiST Declarations and FiST Rules.

2.2 FiST Declarations

FiST declarations change the overall properties of the generated file system. They also define auxiliary data structures that can be used in the rest of the FiST file. The fields of these data structures can be referenced as attributes of the objects they are extending: files, vnodes, ioctls, and mount-time options. We first describe global properties and then data structures. We show possible uses for each.

%accessmode determines if the file system is read-only, write-only, or allows both reading and writing. Read-only file systems can be a useful security feature[23].

%debug determines whether or not debugging code is included. FiST provides extensive debugging support; developers do not have to write debugging statements. If used, our debugging code can trace vnode operations, arguments passed to functions, structures and the values of their fields, etc. Debugging code, when included and not activated, has no noticeable impact on performance. Debugging code, however, when included and activated, can slow performance significantly.

%errorcode defines a new error code to return to user processes. This is used to extend the set of operating system error codes. Developers can use file system specific error codes to specialize their applications. For example, an encryption file system might define EBADKEY to inform a user process that the encryption key used was invalid. (Note that existing user programs can not be aware of new errno codes unless the programs are modified.)

%fanout defines how many file systems will stack immediately under this one. A fan-out of two or more is useful in replicated, load-balancing, unifying[14], or caching file systems[6, 19]. Specifying fan-out values greater than one simplifies code generation because we can deal with a fixed number of lower level vnodes; this also improves performance because we do not have to generate more costly code that handles dynamic fan-out levels. The vnodes of the individual file systems we stack on will be referenced as $1, $2, $3, etc. (See Section 2.3.1.)

%mntstyle determines if mounting this file system will overlay the mounted directory with the mount point. An overlay mount hides the lower level file system. This can be useful in security file systems. For example, our ACL file system (Section 4.3) stores additional access control information in auxiliary files named `.acl`. Aclfs can hide the existence of these ACL files, but would have to use overlay mounting to ensure that the ACL files could not be accessed through the lower level file system. Regular (non-overlay) mounts are useful when fast access to the lower level data is needed. For example, in an encryption file system, a backup utility can backup the data faster (and more securely) by accessing the ciphertext files in the lower level file system.

The following FiST Declarations define data structures that can be used in the rest of the FiST file. The data structures take *basic data types*. Basic data types are those that can be safely copied between user space and the kernel: integers, character arrays, etc.—but not pointers, since these refer to data that resides in two different address spaces.[1]

%fileformat *formatname* {*basic data types*} is used with an auxiliary file as if it were formatted with the data structure. Auxiliary files are used to store additional information about files permanently—information that cannot be stored directly in the files. FiST has commands to read and write auxiliary files with the data in the fields of the data structure as if the data structure were in-memory. Our ACL file system (Section 4.3) uses this feature to store additional access control information in auxiliary files, because we cannot represent it using standard file attributes.

%ioctl declares additional ioctl codes. These ioctls behave like existing ones. They allow users to control the behavior of files or file systems. For example, our encryption file system Section 4.2) uses an ioctl to set cipher keys.

%mntdata defines additional data for user processes to pass once to the kernel during `mount(2)`. For example, a versioning file system can be passed a number indicating the maximum number of versions to allow per file.

%per_vfs defines additional data to store (in memory) for each mounted instance of that file system. This allows developers to define easily a data structure that becomes a part of each mounted file system. For example, a load-balancing file system might flag non-responsive replica servers in a bit array.

%per_vnode defines additional data to store (in memory) per file. This allows developers to define easily a data structure that becomes a part of each file that is in use. The structure's fields automatically become attributes of the file (Section 2.3.1). For example, an encryption file system could use this feature to store per-file cipher keys.

2.3 FiST Rules

This section of a FiST file defines specific actions to perform for file system functions. This is the primary section of the FiST file, where we define important actions. For example, for a load-balancing file system we can write multi-replica lookup instead of the normal lookup; for a versioning file

[1] It is possible to follow user space pointers in the kernel, but we avoided this complexity in the first version of FiST.

system, we can write FiST code to store a backup copy of a file each time it is open for writing.

Figure 4 highlights what a typical stackable vnode operation does: (1) find the vnode of the lower level file system, and (2) repeat the same operation on the lower vnode. The

```
int fsname_getattr(vnode_t *vp, args...)
{
  int error;
  vnode_t *lower_vp = get_lower(vp);

  /* pre-call code goes here */
  /* call same operation on lower file system */
  error = VOP_GETATTR(lower_vp, args...);
  /* post-call code goes here */
  return error;
}
```

Figure 4: Skeleton of typical stackable vnode functions. Pre-call and post-call are explained in Section 2.3.4.

example vnode function receives a pointer to the vnode on which to apply the operation, and other arguments. First, the function finds the corresponding vnode at the lower level file system. Next, the function actually calls the lower level file system through a standard `VOP_*` macro that applies the same operation, but on the file system corresponding to the type of the lower vnode. The macro uses the lower level vnode, and the rest of the arguments unchanged. Finally, the function returns to the caller the status code which the lower level file system passed to the function.

Figure 4 shows two important objects that all vnode functions need to handle: the current vnode (vp) and the corresponding lower vnode (lower_vp). FiST defines variables that refer to arguments passed to the function, such as vnodes, directories to lookup, and file names to create. Variables can refer to the vnode objects at this stack level or those at the lower level. Objects represented by variables can have attributes (e.g., owner, file name, etc.) FiST allows developers to access the attributes of vnodes.

FiST also defines auxiliary functions that abstract kernel functionality that is different across platforms: memory allocation and freeing, string operations, etc. FiST rules can use variables, their attributes, and auxiliary functions.

There are two types of FiST rules: filter rules and vnode function rules. Filter rules simplify the most common and at the same time the most complex actions that a stackable file system can take—manipulating file data and file names. Handling them requires careful coding in many functions, and the code to support changing file names and file data is long and complex. Rather than ask developers to write their code into many vnode functions, we instead centralize this into four functions: two to encode and decode file names, and two to encode and decode file data pages.

The second type of FiST rules are vnode function rules. These rules allow developers to modify the behavior of a single vnode operation or sets of vnode functions collectively. Developers can insert code at the beginning of vnode functions, insert code at the end, or replace the whole code. These rules allow developers to change the function's behavior before accessing the lower level file system, after accessing the lower file system, or change the behavior of the default call to the lower file system.

FiST rules mirror system calls and the NFS protocol for five reasons: First, system calls are familiar to programmers. Second, system calls and the NFS protocol are more standardized than (Unix-specific) vnode interfaces. Third, after comparing several different vnode interfaces, we found that they still act similarly, and that allowed us to translate more popular APIs (system calls and NFS) to each platform's specific vnode interface. Fourth, we plan in the future to port FiST to generate user-level NFS-based file servers. Fifth, by defining a file system API that is as general as possible, we allow future ports of FiST to non-Unix platforms such as Windows NT.

We also allow rules to be applied to sets of vnode functions. For example, in security file systems, developers can apply a rule to all operations that attempt to change file system state (unlink, write, mkdir, etc.), perhaps to detect intruders. Also, an encryption file system that uses per-user keys can insert key validation code at the beginning of all vnode functions—by defining only one rule.

2.3.1 Variables and Attributes

As we already mentioned, FiST variables can refer to vnode objects at this level and the level below us. The form to refer to vnodes and their attributes is

$$\$varg : N.attr \tag{1}$$

where *varg* refers to possible vnode arguments passed to the function, N indicates if we are referring to the current vnode or to one of the lower vnodes, and *attr* refines the specification to one attribute of the vnode. Below are a few examples:

- `$0` is the current or primary vnode
- `$this:0` is the long form of `$0`
- `$2` refers to the second lower vnode of the current vnode in a file system with a fan-out of two or more
- `$dir:0` is the directory vnode of this operation, and may be written more concisely as `$dir` (second form)
- `$1.owner` refers to the user who owns the first (or only) lower vnode
- `$from:2` is the second lower vnode of the source vnode in a rename operation
- `$from:2.name` is the name of the `$from:2` vnode
- `$vfs.key` an encryption key used on this directory, assuming %per_vfs defined "key"
- `$2.fstype` is the name (type) of the second lower file system, when using a fan-out of two ore more

Vnode functions can also access additional data, such as names of files to create (`$name`) and other global variables: %pagesize, %gid, %uid, %pid, %time, and more. Global variables are useful when the file system needs to know credentials of the user accessing the file system, or to find generally fixed properties of the platform (such as native page size).

2.3.2 Auxiliary Functions

FiST defines several useful auxiliary functions that help programmers to write portable code.[2] FiST functions are first class functions. The reason FiST has these functions is to provide a common "library" of functions without having to worry about their internal implementation. For example, different operating systems use different names for their in-kernel memory allocator. Some can allocate kernel memory from different pools, and the type of pool must be specified. When freeing memory, some operating system require the length of buffer to free. FiST abstracts all of this complexity by providing a two simple functions (fistMalloc and fistFree) that work much the same as the ones in standard C libraries.

Other simple auxiliary functions include the following: handling strings (fistStrAdd and fistStrEq, useful when file names are manipulated); buffer copy and comparison functions (fistMemCpy and fistMemCmp); getting and setting errors (fistLastErr, fistSetErr, and fistRetErr); copying files (fistCopyFile); printing messages (fistPrintf); and checking modes and types of files or vnodes (fistIsMode, fistIsDir, fistIsFile, fistIsSymlink, etc.)

All auxiliary functions that take a vnode can also take a file name instead, and FiST will convert the name to its vnode. This makes it easier to write FiST code using either names of existing vnodes to refer to the same file object. Since FiST will open the files if their names were given, it saves a lot of kernel-specific coding from developers.

FiST defines two functions to handle ioctls (fistSetIoctlData and fistGetIoctlData) and two to handle files formatted as data structures (fistSetFileData and fistGetFileData). The *set* functions write into the proper field in an ioctl object or a (persistent) file object the data provided by the developer; the *get* functions retrieve the data from a field in one of these two types of objects. Ioctls are a useful mechanism for user processes to exchange information with the file system (e.g., encryption keys). Files formatted with a given data structure can be used as *auxiliary* files that augment current file system functionality. For example, Aclfs (Section 4.3) stores access controls in special `.acl` files.

Finally, FiST functions can take a variable number of arguments, even allowing developers to skip listing intermediate arguments. Any undefined argument is substituted with suitable defaults. For example, fistRetErr can take zero or one arguments. If an argument is defined, the produced code will return that error from the vnode function. If an argument is not defined, the produced code will return the last error returned from the lower file system.

[2] While FiST provides these auxiliary functions, it also allows users the flexibility to use platform-specific code in FiST files, even if it hinders portability.

2.3.3 Filter Rules and Filter Functions

The most useful and at the same time most complex data manipulations in stackable file system involve file data and file names. To manipulate them consistently without FiST, developers must make careful changes in many places. For example, file data is manipulated in read, write, and all of the MMAP functions; file names also appear in many places: lookup, create, unlink, readdir, etc.

FiST simplifies the task of manipulating file data or file names using two types of *filters*. A filter is a function like Unix shell filters such as sed or sort: they take some input, and produce possibly modified output.

If developers put "`filter data`" in their FiST file, fistgen looks for two data coding functions in the Additional C Code section of the FiST File: *fsname*_encode_data and *fsname*_decode_data. These functions take an input data page, and an allocated output page. Developers are expected to implement these coding functions in the Additional C Code section of the FiST file. The two functions must fill in the page by encoding or decoding it appropriately, and finally return the number of bytes successfully encoded[3]. (Section 2.5 explained why it was simpler to manipulate data in whole pages.) Our encryption file system uses a data filter to encrypt and decrypt data (Section 4.2).

If developers declare "`filter name`" in their FiST file, fistgen inserts code and calls to encode or decode strings representing file names. The file name coding functions (*fsname*_encode_name and *fsname*_decode_name) take an input file name string and its length. They must allocate a new string using fistMalloc, and encode or decode the file name appropriately. Finally, the coding functions return the number of bytes in the newly allocated string, or a negative error code. Fistgen inserts code at the caller's level to free the memory allocated by file name coding functions.

2.3.4 Vnode Function Rules

So far we described FiST's variables and their attributes, how to refer to various vnodes, auxiliary functions, and filter rules. We now turn to the most important part of the FiST rules section: describing how to modify file system operations. Figure 4 showed a typical stackable vnode function and how it breaks into three general parts:

1. **Pre-call**: This part gives the file system a chance to perform some actions before calling the lower level file system. For example, a pre-call code in `unlink` can refuse to delete files whose names end with `.key` because they may contain vital information.
2. **Call**: actually perform the call to the lower file system, passing it the (possibly modified) arguments that were passed to this vnode function. Of the three parts, this is the only mandatory one. A file system that would

[3] Since we do not yet support file systems that change data size (e.g., compression), the number of successfully encoded bytes must be equal to the page size.

allow un-deleting files, for example, could replace the `unlink` call with a `rename` call, essentially saving the file instead of removing it, allowing the file to be restored at a later time.

3. **Post-call**: Post-call code allows developers to define action to take after having called the lower level file system. For example, a lookup in replicated file system may succeed for one replica and fail for another. The file system can return the user a special error code indicating partial failure, the kind that is not severe enough to abort the user program.

The pre-call and post-call parts are normally empty. Users are not required to use them. These two sections allow users the flexibility to perform actions before or after the stackable file system calls the lower level one. The *call* part contains the code to repeat the vnode function on the lower level; users have the option of changing this part.

FiST also lets you apply some operations to sets of file system operations such as all of those that change state (e.g., unlink or write), or all of those that do not change state (e.g., readlink or read). In addition, you can also refer to sets of functions that apply only to file names or to file data. This offers a convenient and concise method of affecting change in many functions at once. For example, an intrusion analysis file system could place post-call code in all of the vnode functions that change state, and log some information that may help it analyze attacks. An encryption file system that wants to authenticate per-user keys can insert the validation code at the beginning of every vnode function, all using a single FiST statement.

The general form for a FiST function rule is:

$$\%callset : optype : part \; \{code\} \tag{2}$$

Callset defines a collection of operations to operate on such as all operations, operations that change state, operations which do not change state, etc. *Optype* further defines the call set to a subset of operations (those that manipulate file names, those that manipulate file data, etc.), or a single operation such as unlink, read, mkdir, symlink, rename, etc. *Part* defines the part of the call that the following code refers to: pre-call, call, or post-call. Finally, *code* contains any C code enclosed in braces. This code is inserted in the proper location in vnode functions and may refer to any FiST variable or function.

2.4 Fistgen

Fistgen is the FiST language code generator. Fistgen reads in an input FiST file, and using the right Basefs templates, produces all the files necessary to build a new file system described in the FiST input file. These output files include C file system source files, headers, sources for user level utilities, and a Makefile to compile them on the given platform.

Fistgen implements a subset of the C language parser and a subset of the C preprocessor. It handles conditional macros (such as #ifdef and #endif). It recognizes the beginning of functions after the first set of declarations and the ending of functions. It parses FiST tags inserted in Basefs (explained in the next section) used to mark special places in the templates. Finally, fistgen handles FiST variables (beginning with $ or %) and FiST functions (such as fistLookup) and their arguments.

After parsing an input file, fistgen builds internal data structures and symbol tables for all the keywords it must handle. Fistgen then reads the templates, and generates output files for each file in the template directory. For each such file, fistgen inserts needed code, excludes unused code, or replaces existing code with another. It also produces several new files (including comments) useful in the compilation for the new file system: a header file for common definitions, and two source files containing auxiliary code.

The auxiliary code generated by fistgen may contain automatically generated functions that are necessary to proper FiST semantics and for FiST functions to be first class. Each FiST function is replaced with one true C function, not a macro, inlined code, a block of code statements, or any feature that may not be portable across operating systems and compilers. While it might have been possible to use other mechanisms such as C macros to handle some of the FiST language, it would have resulted in unmaintainable and unreadable code. One of the advantages of the FiST system is that it produces highly readable code. Developers can edit that code and add more features by hand, if they so choose.

Another compelling reason for fistgen to produce auxiliary functions is that they are sometimes specialized to handle syntax that is not possible in the C language. For example, the fistGetIoctlData function takes arguments that represent names of data structures and names of fields within. A C function cannot pass such arguments; C++ templates would be needed, but we opted against C++ to avoid requiring developers to know another language, because modern Unix kernels are still written in C, and to avoid interoperability problems between C++ produced code and C produced code in a running kernel. Preprocessor macros can handle data structure names and names of fields, but they do not have exact or portable C function semantics. To solve this problem, fistgen replaces calls to functions such as fistGetIoctlData with automatically generated specially named C functions that hard-code the names of the data structures and fields to manipulate. Fistgen generates these auxiliary functions only if needed and only once.

2.5 Basefs

Basefs is a template system similar to Wrapfs[25] and was derived from Wrapfs. Basefs therefore inherited the following design goals from Wrapfs:

- It is a stacking layer that is independent from the layers above and below it. Figure 5 shows this. Basefs appears to the upper VFS as a lower level file system; Basefs also appears to file systems below it as a VFS; all the while, Basefs repeats the same vnode operation on the lower level file system.

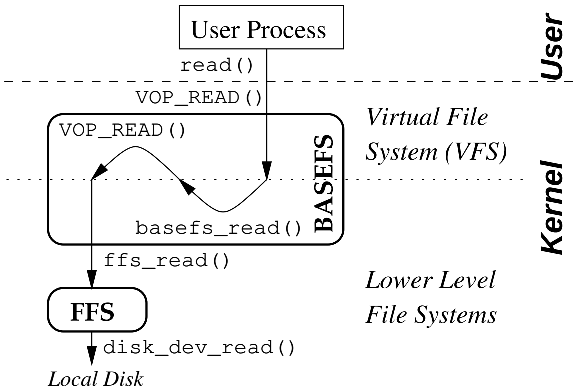

Figure 5: How Basefs Fits Inside the Kernel

- No changes to other file systems, the VFS, or the rest of the kernel are required.[4]
- All data reading and writing operations are performed on whole pages. File systems derived from Basefs manipulate data in whole page chunks and may not change the data size (e.g., compression).
- Since Wrapfs always made copies of data pages, it also cached encoded pages at the lower layer, and cached decoded pages in the upper layer.[5] Basefs improves caching performance by caching at the lower layer only if file data is manipulated and the file system was not overlay mounted. Overlay mounts hide the lower level file system, so direct access to it is impossible, in which case, caching at the lower level would not improve performance and would only consume memory.

Basefs, however, is different from Basefs in several ways. First, substantial portions of code to manipulate file data and file names, as well as debugging code were removed from Basefs. These are included only if the file system needs them. By including only code that is necessary we generate output code that is more readable than code with multi-nested `#ifdef`/`#endif` pairs. Conditionally including this code also resulted in improved performance, as reported in Section 5.3. Matching or beating the performance of other layered file systems was one of the design goals for Basefs.

Second, Wrapfs did not support fan-out file systems natively. Basefs includes that support. This code is also conditionally included, because it is more complex than single-stack file systems, adds more performance overhead, and consumes more memory.

Third, Basefs includes (conditionally compiled) support for many other features that had to be written by hand in Wrapfs. This added support can be thought of as an auxiliary library of common functions: opening, reading or writing, and then closing arbitrary files; overlaying file contents with data structures; user-level utilities to mount and unmount file systems, as well as manipulate ioctls; inspecting and modifying file attributes, and more.

Finally, Basefs includes special *tags* that help fistgen locate the proper places to insert certain code. Inserting code at the beginning or the end of functions is simple, but in some cases the code to add has to go elsewhere. For example, handling newly defined ioctls is done (in the basefs_ioctl vnode function) at the end of a C "switch" statement, right before the "default:" case,

3 Implementation

We implemented the FiST system in Solaris, Linux, and FreeBSD because these three operating systems span the most popular modern Unix platforms and they are sufficiently different from each other. This forced us to understand the generic problems in addition to the system-specific problems. Also, we had access to kernel sources for all three platforms, which proved valuable during the development of our templates. Finally, all three platforms support loadable kernel modules, which sped up the development and debugging process. Loadable kernel modules are a convenience in implementing FiST; they are not required. This section describes the implementation of the two key parts of the FiST system, fistgen and Basefs.

3.1 Implementation of Fistgen

Fistgen translates FiST code into C code. For Unix platforms that support loadable kernel modules, the C code takes the form of a loadable kernel module implementing the file system described in the FiST file. In this section we describe the implementation of key features of FiST that span its full range of capabilities.

We implemented simple global variables such as `%uid` by looking for them in one of the fields from `struct cred` in Solaris or `struct ucred` in FreeBSD. The VFS passes these structures to vnode functions. The Linux VFS simplifies access to credentials by reading that information from the disk inode and into the in-memory vnode structure, `struct inode`. So on Linux we find UID and other credentials by referencing a field directly in the inode which the VFS passes to us.

Most of the vnode attributes listed Section 2.3.1 are simple to find. On Linux they are part of the main vnode structure. On Solaris and FreeBSD, however, we first perform a GETATTR vnode operation to find them, and then return the appropriate field from the structure that the getattr function fills.

The vnode attribute "name" was more complex to implement, because most kernels do not store file names after the initial name lookup routine translates the name to a vnode. On Linux, implementing the vnode name attribute was simple, because it is part of a standard directory entry structure,

[4]Most of our small changes to Linux were incorporated since the 2.3.18 kernel.

[5]Heidemann proposed a solution to the cache coherency problem through a centralized cache manager[4]. His solution, however, required modifications to existing file systems and the rest of the kernel.

`dentry`. On Solaris and FreeBSD, however, we add code to the lookup vnode function that stores the initial file name in the private data of the vnode. That way we could access it as any other vnode attribute, or any other per-vnode attribute added using the `%per_vnode` declaration. To improve performance and reduce kernel memory usage, we include this name storing code only if the FiST file makes use of the vnode name attribute.

Just as we implemented the vnode name attribute as a piece of information in the private data of the vnode, so we implemented all other fields defined using the `%per_vnode` FiST declaration (Section 2.2).

The FiST declarations described in Section 2.2 affect the overall behavior of the generated file system. We implemented the read-only access mode by replacing the call part of every file system function that modifies state (such as unlink and mkdir) to return the error code "read-only file system." We implemented the overlay-only mount style by including code that uses the mounted directory's vnode also as the mount point.

The only difficult part of implementing the `%ioctl` declaration and its associated auxiliary functions, fistGetIoctlData and fistSetIoctlData, was finding how to copy data between user space and kernel space. Solaris and FreeBSD use the routines copyin and copyout; Linux 2.3 uses copy_from_user and copy_to_user.

The last complex feature we implemented was the `%fileformat` FiST declaration and the auxiliary functions used with it: fistGetFileData and fistSetFileData. Consider this small code excerpt:

```
%formatname fmt { data structure }
fistGetFileData(file, fmt, field, out)
```

First, we generate a C data structure named *fmt*. To implement fistGetFileData, we open *file*, read as many bytes from it as the size of the data structure, map these bytes onto a temporary variable of the same data structure type, copy the desired *field* within that data structure into *out*, close the file, and finally return a error/success status value from the function.

Fistgen itself (excluding the templates) is highly portable, and can be compiled on any Unix system. The total number of source lines for fistgen is 4813. Fistgen can process each 1KB of template data in under 0.25 seconds (measured on the same platform used in Section 5.3).

3.2 Implementation of Basefs

The implementation of Basefs proceeded mostly according to its design (Section 2.5). We started with the Wrapfs templates and updated them for newer kernel versions. We then removed debugging code and code to manipulate file names and data pages, because that code gets included conditionally by fistgen only when necessary. We continued by writing support for fan-out file systems, and also wrote some auxiliary support functions (i.e., for reading and writing arbitrary files). Finally, we tagged the Basefs templates in special places where fistgen needs to insert certain code (such as for handling new ioctls).

4 Examples

This section describes the design and implementation of four sample file systems we wrote using FiST. The examples generally progress from those with a simple FiST design to those with a more complex design. Each example introduces a few more FiST features.

1. **Snoopfs**: detects and warns of attempted access to users' files by other non-root users.
2. **Cryptfs**: is an encryption file system.
3. **Aclfs**: adds simple access control lists.
4. **Unionfs**: joins the contents of two file systems.

These examples are experimental and intended to illustrate the kinds of file systems that can be written using FiST. We do not consider them to be complete solutions. We illustrate and discuss only the more important parts of these examples—those that depict key features of FiST. Whenever possible, we mention potential enhancements to our examples. We hope to convince readers of the flexibility and simplicity of writing new file systems using FiST. Additional examples are available elsewhere[22].

4.1 Snoopfs

Snoopfs attempts to detect unauthorized access to users' files. Its premise is that only the file's owner and the root user should be allowed to access users' private files. Anyone else trying (and failing) to access them, may be considered an intruder. Snoopfs shows the use of one of the more useful file system functions, lookup. The FiST code for Snoopfs uses only one rule in FiST Rules section:

```
%op:lookup:postcall {
if ((fistLastErr() == EPERM ||
     fistLastErr() == ENOENT) &&
    $0.owner != %uid && %uid != 0)
  fistPrintf("snoopfs detected access by uid %d,\
pid %d, to file %s\n", %uid, %pid, $name);
}
```

If the accessing user is neither root nor the file's owner, and the error from looking up the file was either "permission denied" or "no such entry," then we print a warning message to the console. The message may be logged via `syslog(3)` (which also adds a timestamp). Note that successful access to world-readable or group-readable files will not result in a warning.

4.2 Cryptfs

Cryptfs is a strong encryption file system. It uses the Blowfish[18] encryption algorithm in Cipher Feedback (CFB) mode[17]. We used one fixed Initialization Vector

(IV), and one 128-bit key per mounted instance of Cryptfs. Cryptfs encrypts both file data and file names. After encrypting file names, Cryptfs also uuencodes them to avoid characters that are illegal in file names. Additional design and important details are available elsewhere[24].

The FiST implementation of Cryptfs shows three additional features: file data encoding, using ioctl calls, and using per-VFS data. Cryptfs's FiST code uses all four sections of a FiST file. Most of the code for Cryptfs is:

```
%{
#include <blowfish.h>
%}
filter:data;
filter:name;
ioctl:fromuser SETKEY {char ukey[16];};
per_vfs {char key[16];};
%%
%op:ioctl:SETKEY {
  char temp_buf[16];
  if (fistGetIoctlData(SETKEY, ukey, temp_buf) < 0)
    fistSetErr(EFAULT);
  else
    BF_set_key(&$vfs.key, 16, temp_buf);
}
%%
unsigned char global_iv[8] = {
  0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10 };
int cryptfs_encode_data(const page_t *in,
                        page_t *out)
{
  int n = 0;  /* blowfish variables */
  unsigned char iv[8];

  fistMemCpy(iv, global_iv, 8);
  BF_cfb64_encrypt(in, out, %pagesize,
                   &($vfs.key), iv, &n,
                   BF_ENCRYPT);
  return %pagesize;
}
```

We omitted the call to decode data and the calls to encode and decode file names because they are similar in behavior to data encoding. Cryptfs defines an ioctl named SETKEY, used to set 128-bit encryption keys. We wrote a simple user-level tool which prompts the user for a passphrase and sends an MD5-hash of it to the kernel using this ioctl. When the SETKEY ioctl is called, Cryptfs stores the (cipher) key in the private VFS data field "key", to be used later.

There are several possible extensions to Cryptfs: storing per-file or per-directory keys in auxiliary files that would otherwise remain hidden from users' view, much the same as Aclfs does (Section 4.3.); using several types of encryption algorithms, and defining mount flags (using `%mntflag`) to select among them.

4.3 Aclfs

Aclfs allows an additional UID and GID to share access to a directory as if they were the owner and group of that directory. Aclfs shows three additional features of FiST: using the more secure overlay mounts, using special purpose auxiliary files, and hiding files from users' view. The FiST code for Aclfs uses the FiST Declarations and FiST Rules sections:

```
mntstyle overlay;
ioctl:fromuser SETACL int u; int g;;
fileformat ACLDATA int us; int gr;;
%%
%op:ioctl:SETACL {
  if ($0.owner == %uid) {
    int u2, g2;
    if (fistGetIoctlData(SETACL, u, &u2) < 0 ||
        fistGetIoctlData(SETACL, g, &g2) < 0)
      fistSetErr(EFAULT);
    else {
      fistSetFileData(".acl", ACLDATA, us, u2);
      fistSetFileData(".acl", ACLDATA, gr, g2);
    }
  } else
    fistSetErr(EPERM);
}
%op:lookup:postcall {
  int u2, g2;
  if (fistLastErr() == EPERM &&
      fistGetFileData(".acl", ACLDATA, us, u2)>=0 &&
      fistGetFileData(".acl", ACLDATA, gr, g2)>=0 &&
      (%uid == u2 || %gid == g2))
        fistLookup($dir:1, $name, $1,
                   $dir:1.owner, $dir:1.group);
}
%op:lookup:precall {
  if (fistStrEq($name, ".acl") &&
      $dir.owner != %uid)
    fistReturnErr(ENOENT);
}
%op:readdir:call {
  if (fistStrEq($name, ".acl"))
    fistSkipName($name);
}
```

When looking up a file in a directory, Aclfs first performs the normal access checks (in `lookup`). We insert postcall code after the normal lookup that checks if access to the file was denied and if an additional file named `.acl` exists in that directory. We then read one UID and GID from the `.acl` file. If the effective UID and GID of the current process match those listed in the `.acl` file, we repeat the lookup operation on the originally looked-up file, but using the ownership and group credentials of the *actual* owner of the directory. We must use the owner's credentials, or the lower file system will deny our request.

The `.acl` file itself is modifiable only by the directory's owner. We accomplish this by using a special ioctl. Finally, we hide `.acl` files from anyone but their owner. We insert code in the beginning of lookup that returns the error "no such file" if anyone other than the directory's owner attempted to lookup the ACL file. To complete the hiding of ACL files, we skip listing `.acl` files when reading directories.

Aclfs shows the full set of arguments to the fistLookup routine. In order, the five arguments are: the directory to lookup in, the name to lookup, the vnode to store the newly looked up entry, and the credentials to perform the lookup with (UID and GID, respectively).

There are several possible extensions to this implementation of Aclfs. Instead of using the UID and GID listed in the `.acl` file, it can contain an arbitrarily long list of user and group IDs to allow access to. The `.acl` file may also include sets of permissions to deny access from, perhaps using negative integers to distinguish them from access permissions. The granularity of Aclfs can be made on a per-file basis; for each file F, access permissions can be read from a file `.F.acl`, if one exists.

4.4 Unionfs

Unionfs joins the contents of two file systems similar to BSD-4.4's Unionfs[14]. The two lower file systems can be considered two branches of a stackable file system tree. Unionfs shows how to merge the contents of directories in FiST, and how to define behavior on a set of file system operations. The FiST code for Unionfs uses the FiST Declarations and FiST Rules sections:

```
fanout 2;
%%
%op:lookup:postcall {
  if (fistLastErr() == ENOENT)
    fistSetErr(fistLookup($dir:2, $name));
}
%op:readdir:postcall {
  fistSetErr(fistReaddir($dir:2, NODUPS));
}
%delops:all:postcall {
   fistSetErr(fistOp($2));
}
%writeops:all:call {
  fistSetErr(fistOp($1));
}
```

Normal lookup will try the first lower file system branch ($1). We add code to lookup in the second branch ($2) if the first lookup did not find the file. If a file exists in both lower file systems, Unionfs will use the one from the first branch. Normal directory reading is augmented to include the contents of the second branch, but setting a flag to eliminate duplicates; that way files that exist in both lower file systems are listed only once. Since files may exist in both branches, they must be removed (unlink, rmdir, and rename) from all branches. Finally we declare that all writing operations should perform their respective operations only on the first branch; this means that new files are created in the first branch where they will be found first by subsequent lookups.

There are several other issues file system semantics and especially concerning error propagation and partial failures, but these are beyond the scope of this paper. Extensions to our Unionfs include larger fan-outs, masking the existence of a file in $2 if it was removed from $1, and ioctls or mount options to decide the order of lookups and writing operations on the individual file system branches.

5 Evaluation

We evaluate the effectiveness of FiST using three criteria: code size, development time, and performance. We show how development and porting times are dramatically reduced when using FiST. We also show that performance overhead is small and comparable to other stacking work. We report results based on the four example file systems described in Section 4, Snoopfs, Cryptfs, Aclfs, and Unionfs, on three different platforms: Linux 2.3, Solaris 2.6, and FreeBSD 3.3.

5.1 Code Size

Code size is one measure of the development effort necessary for a file system. To demonstrate the savings in code size achieved using FiST, we compare the number of lines of code that need to be written to implement the four example file systems in FiST versus three other implementation approaches: writing C code using a standalone version of Basefs, writing C code using Wrapfs, and writing the file systems from scratch as kernel modules using C. In particular, we first wrote all four of the example file systems from scratch before writing them using FiST. For these example file systems, the C code generated from FiST was identical in size to the hand-written code.

When counting lines of code, we excluded comments, empty lines, and %% separators. For Cryptfs we excluded 627 lines of C code of the Blowfish encryption algorithm, since we did not write it. When counting lines of code for implementing the example file systems using the Basefs and Wrapfs stackable templates, we exclude code that is part of the templates and only count code that is specific to the given example file system. These results are shown in Table 1 for Linux 2.3, Solaris 2.6, and FreeBSD 3.3. For reference, we include the code sizes of Basefs and Wrapfs and also show the number of lines of code required to implement Wrapfs in FiST and Basefs. Table 1 shows huge reductions in code size when comparing FiST versus writing hand-written code from scratch. We focus though on the comparison of FiST versus stackable template systems. As Wrapfs represents the most conservative comparison, the table shows for each file system the reduction factor in the number of lines of code written using Wrapfs compared to FiST. The average code size reduction in using FiST versus Wrapfs across all four file systems on all four platforms is a factor of 10. Table 1 shows three size reduction classes:

1. For simple file systems such as Snoopfs (on Linux) and Cryptfs, code size was reduced by 30–50%. This reduction factor is relatively small because the generated code matches almost one-to-one with the FiST code. The reduction factor for Snoopfs on Solaris and FreeBSD is 5–6 because on those two platforms, finding the owner of a file requires calling the GETATTR vnode operation, while on Linux this information is part of the in-memory inode structure.
2. Moderate savings (5–6 times) are achieved for Aclfs and for Snoopfs on Solaris and FreeBSD. The reason for this is that some lines of FiST code for these file systems produce ten or more lines of C code.
3. The largest savings appeared for Unionfs, a factor of 28–33 times. The reason for this is that fan-out file systems produce C code that affects all vnode operations; each vnode operation must handle more than one lower

Op. Sys.	File System	FiST Code	C code given		Wrapfs / FiST	From Scratch
			Basefs	Wrapfs		
Linux 2.3	Basefs	0	0	0		2594
	Wrapfs	40	772	0	0.0	3366
	Snoopfs	4	6	6	1.5	2600
	Cryptfs	99	896	124	1.3	3513
	Aclfs	39	214	214	5.5	2835
	Unionfs	15	500	500	33.3	3094
Solaris 2.6	Basefs	0	0	0		2914
	Wrapfs	40	1034	0	0.0	3948
	Snoopfs	4	20	20	5.0	2934
	Cryptfs	99	1158	124	1.3	4072
	Aclfs	39	224	224	5.7	3138
	Unionfs	15	420	420	28.0	3334
FreeBSD 3.3	Basefs	0	0	0		2751
	Wrapfs	40	1693	0	0.0	4058
	Snoopfs	4	24	24	6.0	2755
	Cryptfs	99	1817	124	1.3	4182
	Aclfs	39	216	216	5.5	2967
	Unionfs	15	481	481	32.1	3232

Table 1: Number of additional lines of code for various file systems, over that of Basefs or Wrapfs, and the size reduction factor when using FiST. The last column counts the total number of lines written in C from scratch.

vnode. This additional code was not part of the original Wrapfs implementation, and it is not used unless fan-outs of two or more are defined (to save memory and improve performance). If we exclude the code to handle fan-outs, Unionfs's added C code is still over 100 lines producing savings of a factor of 7–10.[6]

Table 1 shows the code sizes for *each* platform. The savings gained by FiST are leveraged with each port. If we sum up the savings for the above three platforms, we reach reduction factors ranging from 4 to over 100 times. The more ports of fistgen exist, the better these cumulative savings would be.

5.2 Development Time

Estimating the time to develop kernel software is very difficult. Developers' experience can affect this time significantly, and this time is generally reduced with each port. In this section we report our own personal experiences given these file system examples and the three platforms we worked with. Table 2 shows the number of days we spent developing various file systems and porting them to four different platforms.

We estimated the incremental time spent developing each file system, assuming 8 hour work days, and using our source commit logs and change logs. We estimated the time it took us to develop Wrapfs, Basefs, and the example file systems. Then we measured the time it took us to develop each of these file systems using the FiST language.

[6]FreeBSD's Unionfs is 4863 lines long, which is 50% larger than our Unionfs (3232 lines). FreeBSD's Unionfs is 2221 lines longer than their Nullfs, while ours is only 481 lines longer than our Basefs. Unfortunately, the stacking infrastructure in FreeBSD is currently broken, so we were unable to compare the performance of our stacking to FreeBSD's.

File System	In FiST	Time given		Wrapfs / FiST	From Scratch
		Basefs	Wrapfs		
Basefs	0	0	0		48
Wrapfs	0.1	22	0	0.0	70
Snoopfs	0.1	0.5	0.5	5.0	8.5
Cryptfs	2	25	2	1.0	73
Aclfs	0.2	3	3	15.0	48.5
Unionfs	1	2	7	7.0	55

Table 2: Average number of days per platform to develop various file systems using C or using FiST, the increment given Basefs or Wrapfs, and the time reduction factor.

For most file systems, incremental time savings are a factor of 5–15 because hand writing C code for each platform can be time consuming, while FiST provides this as part of the base templates and the additional library code that comes with Basefs. For Cryptfs, however, there are no time savings per platform, because the vast majority of the code for Cryptfs is in implementing the four encoding and decoding functions, which are implemented in C code in the Additional C Code section of the FiST file. The average per platform reduction in development time across the four file systems is a factor of seven in using FiST versus Wrapfs templates.

When taking into account multiple platforms, the time savings become more significant, since FiST code is written only once. Given our three platforms, accumulated savings range from 3–45 times.

5.3 Performance

To evaluate the performance of file systems written using FiST, we tested each of the example file systems by mounting it on top of a disk based native file system and running benchmarks in the mounted file system. We conducted measurements for Linux 2.3, Solaris 2.6, and FreeBSD 3.3. The native file systems used were ext2, ufs, and ffs, respectively. We measured the performance of our file systems using two methods: (1) compiling a large package (am-utils-6.0, 50,000 lines of C code), and (2) micro-benchmarks intended to isolate the impact of each file system. The micro-benchmarks included a series of recursive copies (cp -r), recursive removals (rm -rf), recursive find, and "find-grep" (find /mnt -print | xargs grep *pattern*). Each benchmark was run once to warm up the cache, after which we took 10 new measurements and averaged them. The standard deviation for our measurements was less than 2% of the mean. We ran all tests on the same machine: a P5/90, 64MB RAM, and a Quantum Fireball 4.35GB IDE hard disk.

The most important performance metric is the basic overhead imposed by our templates. The overhead of Basefs over the file systems it mounts on is just 0.8–2.1%. This minimum overhead is below the 3–10% degradation previously reported for null-layer stacking[6, 19].

Compared to Basefs, Wrapfs also manipulates file names and file data. These extra data copies add another 4.2–4.9% overhead over Basefs. This added overhead was part of the Wrapfs, but is not incurred in Basefs unless the file systems derived from it require file data or file name manipulations.

The performance of individual file systems can vary greatly depending on the operating system in question. Compared to Basefs, the performance of Snoopfs on Linux is less than 1%; on Solaris and FreeBSD, Snoopfs adds another 5% overhead. The reason for the latter is that Snoopfs has to perform an additional LOOKUP operation each time it has to find the owner of a file (evaluating $0.owner in Section 4.1). Running this additional LOOKUP is the only VFS API on these two platforms that can provide this information. On Linux, however, finding the owner is a simple dereferencing of a field in the inode structure, because the Linux VFS provides such information automatically.

The performance of individual file systems can also vary much depending on the actual actions defined in the FiST Rules and Additional C Code sections, since they allow developers to write arbitrarily complex code. For example, the overhead added by Cryptfs over Basefs is 9.1–22.9%, and is due substantially to the cost of the Blowfish encryption[24, 25]. When testing Aclfs, we ran half of the tests as the owner of the test directories, and the rest as a user authorized (in the `.acl` file) to access these directories. Aclfs adds an overhead of 3–5% over Basefs, more than 80% of which is incurred when reading `.acl` files and repeating lookups. To benchmark Unionfs using the large compile benchmark, we copied half of the files in the am-utils package to the first branch, and the all of them to the second branch. That way about half the files would have to be looked up a second time (after not being found on the first branch). Unionfs's overhead over Basefs is 3.2–7.7%.

Finally, since we did not change the VFS, and all of our stacking work is in the templates, there is no overhead on the rest of the system; performance of native file systems (NFS, FFS, etc.) is unaffected when our stacking is not used.

6 Related Work

Rosenthal first implemented stacking in SunOS 4.1 in the early 1990s[15]. A few other works followed his, such as further prototypes for extensible file systems in SunOS[19], and the Ficus layered file system[3, 5]. Webber implemented file system interface extensions that allow user-level file servers[21]. Unfortunately, these implementations required modifications to either existing file systems or the rest of the kernel, limiting their portability significantly, and affecting the performance of native file systems. FiST achieves portability using a minimal stackable base file system, Basefs, which can be ported to another platform in 1–3 weeks. No changes need to be made to existing kernels or file systems, and there is no performance penalty for native file systems.

Newer Unix operating systems, such as the HURD[2], Spring[11], and the Exokernel[8] have an extensible file system interface. The HURD is a set of servers running under the Mach 3.0 microkernel[1] that collectively provide a Unix-like environment. HURD translators are programs that can be attached to a pathname and perform specialized services when that pathname is accessed. Writing translators entails implementing a well defined file access interface and filling in stub operations for reading files, creating directories, listing directory contents, etc.

Sun Microsystems Laboratories built Spring, an object-oriented research operating system[11]. Spring was designed as a set of cooperating servers on top of a microkernel. It provides generic modules that offer services useful for a file system: caching, coherency, I/O, memory mapping, object naming, and security. Writing a file system for Spring involves defining the operations to be applied on the objects. Operations not defined are inherited from their parent object. One work that resulted from Spring is the Solaris MC (Multi-Computer) File System[10]. It borrowed the object-oriented interfaces from Spring and integrated them with the existing Solaris vnode interface to provide a distributed file system infrastructure through a special *Proxy File System*. Solaris MC provides all of Spring's benefits, while requiring little or no change to existing file systems; those can be ported gradually over time. Solaris MC was designed to perform well in a closely coupled cluster environment (not a general network) and requires high performance networks and nodes.

The Exokernel is an extensible operating system that comes with XN, a low-level in-kernel stable storage system[8]. XN allows users to describe the on-disk data structures and the methods to implement them (along with file system libraries called libFSes). The Exokernel requires significant porting work to each new platform, but then it can run many unmodified applications.

The main disadvantages of the HURD, Spring, and the Exokernel are that they are not portable enough, not sufficiently developed or stable, or they are not available for general use. In comparison, FiST provides portable stacking on widely available operating systems. Finally, none of the related extensible file systems come with a high-level language that developers can use to describe file systems.

7 Conclusions

The main contribution of this work is the FiST language which can describe stackable file systems. From a single FiST description we generate code for different platforms. We achieved this portability because FiST uses an API that combines common features from several vnode interfaces. FiST saves its developers from dealing with many kernel internals, and lets developers concentrate on the core issues of the file system they are developing. FiST reduces the learning curve involved in writing file systems, by enabling non-experts to write file systems more easily.

The most significant savings FiST offers is in reduced development and porting time. The average time it took us to develop a stackable file system using FiST was at least seven times faster than when we wrote the code using Basefs. These savings multiple for each platform to which FiST is ported. We showed how FiST descriptions are more concise than hand-written C code: 5–8 times smaller for average stackable file systems, and as much as 33 times smaller for more complex ones. FiST generates file system modules that run in the kernel, thus benefiting from increased performance over user level file servers. The minimum overhead imposed by our stacking infrastructure is 1–2%.

FiST can be ported to other Unix platforms in 1–3 weeks, assuming the developers have access to kernel sources. The benefits of FiST are increased each time it is ported to a new platform: existing file systems described with FiST can be used on the new platform without modification.

7.1 Future Work

The FiST language is fairly portable, and its API maps well to NFS's protocol calls[13]. We plan for fistgen to generate user-level NFS-based code, using one of several user-level NFS servers as a base template. While user-level file servers are slower, they are more portable and easier to debug.

We also plan to port our system to Windows NT. NT has a different file system interface than Unix's vnode interface. NT's I/O subsystem defines its file system interface. NT *Filter Drivers* are optional software modules that can be inserted above or below existing file systems[12]. Their task is to intercept and possibly extend file system functionality. One example of an NT filter driver is its virus signature detector. It is possible to emulate file system stacking under NT. Our estimates are that porting Basefs to NT is possible, but will take 2–3 months, not 1–3 weeks as we predict for Unix ports.

Other features we plan to support include the ability to handle file systems that change file sizes (compression) and be able to specify more complex data structures and lists of structures in FiST auxiliary files. Finally, we are exploring layer collapsing in FiST: a method to generate one file system that merges the functionality from several FiST descriptions, thus saving the per-layer stacking overheads.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *USENIX Conf. Proc.*, pages 93–112, 1986.

[2] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU's Bulletin*. Free Software Foundation, 1994. Copies are available by writing to `gnu@prep.ai.mit.edu`.

[3] 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 Conf. Proc.*, pages 63–71, 1990.

[4] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Fifteenth ACM SOSP*. ACM SIGOPS, 1995.

[5] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Tech-report CSD-910007. UCLA, 1991.

[6] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *ACM ToCS*, **12**(1):58–89, Feb., 1994.

[7] S. C. Johnson. *Yacc: Yet Another Compiler-Compiler*, UNIX Programmer's Manual Volume 2 — Supplementary Documents. Bell Laboratories, Murray Hill, New Jersey, July 1978.

[8] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. *Sixteenth ACM SOSP*, pages 52–65, 1997.

[9] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *USENIX Conf. Proc.*, pages 238–47, 1986.

[10] V. Matena, Y. A. Khalidi, and K. Shirriff. Solaris MC File System Framework. Tech-report TR-96-57. Sun Labs, 1996. http://www.sunlabs.com/technical-reports/1996/abstract-57.html.

[11] J. G. Mitchell, J. J. Gibbons, 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 Conf. Proc.*, 1994.

[12] R. Nagar. Filter Drivers. In *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, 1997.

[13] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. *USENIX Conf. Proc.*, pages 137–52, 1994.

[14] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. *USENIX Conf. Proc. on UNIX and Advanced Computing Systems*, pages 25–33, 1995.

[15] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conf. Proc.*, pages 107–18. USENIX, 1990.

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

[17] B. Schneier. Algorithm Types and Modes. In *Applied Cryptography, 2nd ed.*, pages 189–97. John Wiley & Sons, 1996.

[18] B. Schneier. Blowfish. In *Applied Cryptography, Second Edition*, pages 336–9. John Wiley & Sons, 1996.

[19] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. *USENIX Conf. Proc.*, pages 161–74, 1993.

[20] SMCC. lofs – loopback virtual file system. SunOS 5.5.1 Reference Manual, Section 7. Sun Microsystems, Inc., 20 March 1992.

[21] N. Webber. Operating System Support for Portable Filesystem Extensions. *USENIX Conf. Proc.*, pages 219–25, 1993.

[22] E. Zadok. The FiST Home Page. http://www.cs.columbia.edu/~ezk/research/fist/.

[23] E. Zadok. Stackable File Systems as a Security Tool. Work in Progress. Computer Science Department, Columbia University, 1999.

[24] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 1998.

[25] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. *USENIX Conf. Proc.*, 1999.

Software and additional documentation is available in http://www.cs.columbia.edu/~ezk/research/fist.