FiST: A Language for Stackable File Systems

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

Abstract

Traditional file system development is difficult. Stackable file systems promise to ease the development of file systems by offering a mechanism for incremental development. Unfortunately, existing methods often require writing complex low-level kernel code that is specific to a single operating system platform and also difficult to port.

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

File systems have proven to be useful in enriching system functionality. The abstraction of folders with files containing data is natural for use with existing file browsers, text editors, and other tools. Modifying file systems became a popular method of extending new functionality to users. However, developing file systems is difficult and involved. Developers often use existing code for native in-kernel file systems as a starting point[15, 23]. Such file systems are difficult to write and port because they depend on many operating system specifics, and they often contain many lines of complex operating systems code, as seen in Table 1.

Media Type	**Common File System**	**Avg. Code Size** (C lines)
Hard Disks	UFS, FFS, EXT2FS	5,000–20,000
Network	NFS	6,000–30,000
CD-ROM	HSFS, ISO-9660	3,000–6,000
Floppy	PCFS, MS-DOS	5,000–6,000

Table 1: Common Native Unix File Systems and Code Sizes for Each Medium

User-level file systems are easier to develop and port because they reside outside the kernel[16]. However, their performance is poor due to the extra context switches these file systems must incur. These context switches can affect performance by as much as an order of magnitude[26, 27].

Stackable file systems[19] promise to speed file system development by providing an extensible file system interface. This extensibility allows new features to be added incrementally. Several new extensible interfaces have been proposed and a few have been implemented[8, 15, 18, 22]. To improve performance, these stackable file systems were designed to run in the kernel. Unfortunately, using these stackable interfaces often requires writing lots of complex C kernel code that is specific to a single operating system platform and also difficult to port.

More recently, we introduced a stackable template system called Wrapfs[27]. It eases up file system development by providing some built-in support for common file system activities. It also improves portability by providing kernel templates for several operating systems. While working with Wrapfs is easier than with other stackable file systems, developers still have to write kernel C code and port it using the platform-specific templates.

In previous approaches, performance and portability could not be achieved together. To perform well, a file system should run in the kernel, not at user level. Kernel code, however, is much more difficult to write and port than user-level code. To ease the problems of developing and porting stackable file systems that perform well, we propose a high-level language to describe such file systems. There are three benefits to using a language:

1. **Simplicity:** A file system language can provide familiar higher-level primitives that simplify file system development. The language can also define suitable defaults automatically. These reduce the amount of code that developers need to write, and lessen their need for extensive knowledge of kernel internals, allowing even non-experts to develop file systems.
2. **Portability:** A language can describe file systems using an interface abstraction that is common to operating systems. The language compiler can bridge the gaps among different systems' interfaces. From a single description of a file system, we could generate file system code for different platforms. This improves portability considerably. At the same time, however, the language should allow developers to take advantage of system-specific features.
3. **Specialization:** A language allows developers to customize the file system to their needs. Instead of having one large and complex file system with many features that may be configured and turned on or off, the compiler can produce special-purpose file systems. This improves performance and memory footprint because specialized file systems include only necessary code.

This paper describes the design and implementation of *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. With FiST, developers need only describe the core functionality of their file systems. The FiST language code generator, *fistgen*, generates kernel file system modules for several platforms using a single description. We currently support Solaris, FreeBSD, and Linux.

To assist fistgen with generating stackable file systems, we created a minimal stackable file system template called Basefs. Basefs adds stacking functionality missing from systems and relieves fistgen from dealing with many platform-dependent aspects of file systems. Basefs does not require changes to the kernel or existing file systems. Its main function is to handle many kernel details relating to stacking. 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 several 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

FiST is a high level language providing a file system abstraction. Figure 1 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 are stackable file systems such as the examples in Section 4, as well as Basefs. These file systems provide a higher abstraction than native file systems because stackable file systems interact only with other file systems through a well defined *virtual file system interface* (VFS)[11]. The VFS provides *virtual nodes* (vnodes), an abstraction of files across different file systems. However, both these levels are system specific.

<table>
<tr><th colspan="2">FiST Language</th></tr>
<tr><td>Basefs templates</td><td>Stackable (VFS) File Systems
(lofs, cryptfs, aclfs, unionfs, etc.)</td></tr>
<tr><td colspan="2">Low-Level File Systems (UFS, NFS, etc.)</td></tr>
</table>

Figure 1: 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.

At the highest level, we define the FiST language. FiST abstracts the different vnode interfaces across *different* operating systems into a single common description language, because it is easier to write file systems this way. We found that while vnode 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: we mirror existing platform-specific vnode interfaces, and extend them through the FiST language in a platform independent way. This allows us to modify vnode operations and the arguments they pass in an arbitrary way, providing great design flexibility. At the same time, this abstraction means that stackable file systems cannot easily access device drivers and control, for example, block layout of files on disks and the existing structure of meta-data (inodes).

FiST does not require that applications be changed. The default behavior of produced code maintains compatibility with existing file system APIs. FiST does allow, however, the creation of special-purpose file systems that can extend new functionality to applications.

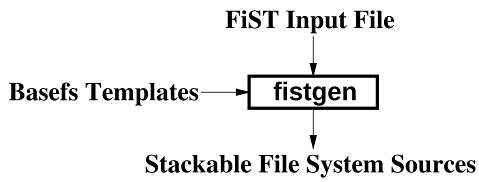

Figure 2: FiST Operational Diagram. Fistgen reads a FiST input file, and with the Basefs templates, produces sources for a new file system.

The overall operation of the FiST system is shown in Figure 2. The figure illustrates how the three parts of FiST work 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 then 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 kernel C sources as output. The latter 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.

2.1 A Quick Example: Snoopfs

To illustrate the FiST development process, we contrast it with traditional file system development methods using a simple example similar to Watchdogs[2]. Suppose a file system developer wants to write a file system that will warn of any possible unauthorized access to users' files. The main idea is that only the files' owner or the root user are allowed access. Any other user who might be attempting to find files that belong to another user, would normally get a "permission denied" error code. However, the system does not produce an alert when such an attempt is made. This new *snooping file system* (Snoopfs) will log these failed attempts.

The one place where such a check should be made is in the lookup routine that is used to find a file in a directory. To do so without FiST, the developer has to do the following:

1. locate an operating system with available sources for one file system
2. read and understand the code for that file system and any associated kernel code
3. make a copy of the sources, and carefully modify them to include the new functionality
4. compile the sources into a new file system, possibly rebuilding a new kernel and rebooting the system
5. mount the new file system, test, and debug as needed

After completing this, the developer is left with one modified file system for one operating system. The amount of code that has to be read and understood ranges in the thousands of lines (Table 1). The process has to be repeated for each new port to a new platform. In addition, changes to native file system are unlikely to be accepted by operating system maintainers, and have to be maintained independently.

In contrast, the normal procedure for developing code with FiST is:

1. write the code in FiST once
2. run fistgen on the input file
3. compile the produced sources into a loadable kernel module, and load it into a running system
4. mount the new file system, test, and debug as needed

Debugging code can be turned on in FiST to assist in the development of the new file system. There is no need to have kernel sources or be familiar with them; there is no need to write or port code for each platform; and there is no need to rebuild or reboot the kernel. Furthermore, the same developer can write Snoopfs using a small number of lines of FiST code:

```
%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);
}
```

This short FiST code inserts an "if" statement after the normal call to the lookup routine. The code checks if the previous lookup call failed with one of two particular errors, who the owner of the directory is, who the effective running user is, and then decides whether to print the warning message.

This single FiST description is portable, and can be compiled on each platform that we have ported our templates to (currently three).

2.2 The File System Model

A FiST-produced file system runs in the kernel, as seen in Figure 3. FiST file systems mirror the vnode interface both above and below. The interface to user processes is the system call interface. FiST does not change either the system call interface or the vnode interface. Instead, FiST can change information passed and returned through the interfaces.

A user process generally accesses a file system by executing a system call, which traps into the kernel. The kernel VFS then translates the system call to a vnode operation, and calls the corresponding file system. If the latter is a FiST-produced file system, it may call another stacked file system below. Once the execution flow has reached the lowest file system, error codes and return values begin flowing upwards, all the way to the user process.

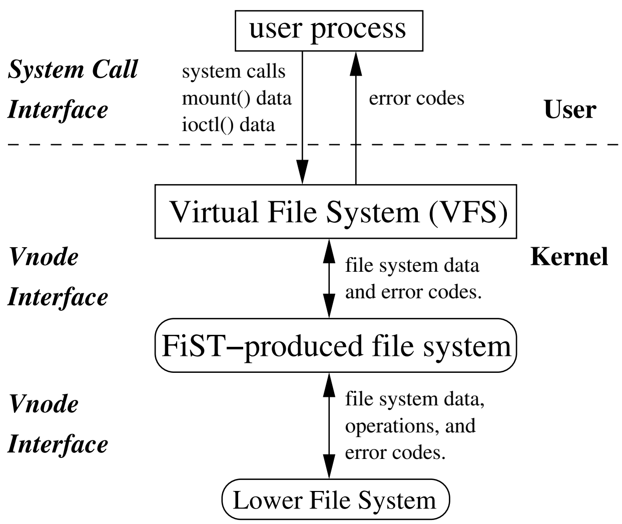

Figure 3: Information and execution flow in a stackable system. FiST does not change the system call or vnode interfaces, but allows for arbitrary data and control operations to flow in both directions.

In FiST, we model a file system as a collection of mounts, files, and user processes, all running under one system. Several *mounts*, mounted instances of file systems, can exist at any time. A FiST-produced file system can access and manipulate various mounts and files, data associated with them, their attributes, and the functions that operate on them. Furthermore, the file system can access attributes that correspond to the run-time execution environment: the operating system and the user process currently executing.

Information (both data and control) generally flows between user processes and the mounted file system through the system call interface. For example, file data flows between user processes and the kernel via the `read` and `write` system calls. Processes can pass specific file system data using the `mount` system call. In addition, mounted file systems may return arbitrary (even new) error codes back to user processes.

Since a FiST-produced stackable file system is the caller of other file systems, it has a lot of control over what transpires between it and the ones below through the vnode interface. FiST allows access to multiple mounts and files. Each mount or file may have multiple attributes that FiST can access. Also, FiST can determine how to apply vnode functions on each file. For maximum flexibility, FiST allows the developer full control over mounts and files, their data, their attributes, and the functions that operate on them; they may be created or removed, data and attributes can be changed, and functions may be augmented, replaced, reordered, or even ignored.

Ioctls (I/O Controls) have been used as an operating system extension mechanism as they can exchange arbitrary information between user processes and the kernel, as well as in between file system layers, without changing interfaces. FiST allows developers to define new ioctls and define the actions to take when they are used; this can be used to create application-specific file systems. FiST also provides functions for portable copying of ioctl data between user and kernel spaces. For example, our encryption file system (Section 4.1) uses an ioctl to set cipher keys.

Traditional stackable file systems create a single linear stack of mounts, each one hiding the one file system below it. More general stacking allows for a tree-like mount structure, as well as for direct access to any layer[8, 18]. This interesting aspect of stackable file systems is called fanning, as shown in Figure 4. A *fan-out* allows the mounted file system to access two or more mounts below. A fan-out is useful for example in replicated, load-balancing, unifying[15], or caching file systems[22].

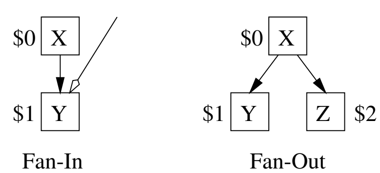

Figure 4: Fanning in stackable file systems

A *fan-in* allows a process to access lower level mounts directly. This can be 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. If fan-in is not used, the mounted file system will overlay the mounted directory with the mount point. An overlay mount hides the lower level file system. This can be useful for some security applications. For example, our ACL file system (Section 4.2) hides certain important files from normal view and is able to control who can manipulate those files and how.

2.3 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 offer 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 5 shows the four main sections of a FiST input file.

1	`%{`
	C Declarations
	`%}`

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

Figure 5: FiST Grammar Outline

The FiST grammar was modeled after yacc[9] input files, because yacc is familiar to programmers and the purpose for each of its four sections (delimited by "%%") matches with four different subdivisions of desired file system code: raw included header declarations, declarations that affect the produced code globally, actions to perform when matching vnode operations, and additional code.

C Declarations (enclosed in "{% %}") are used to include additional C headers, define macros or typedefs, list forward function prototypes, etc. These declarations are used throughout the rest of the code.

FiST Declarations define global file system properties that affect the overall semantics of the produced code and how a mounted file system will behave. These properties are useful because they allow developers to make common global changes in a simple manner. In this section we declare if the file system will be read-only or not, whether or not to include debugging code, if fan-in is allowed or not, and what level (if any) of fan-out is used.

FiST Declarations can also define special data structures used by the rest of the code for this file system. We can define mount-time data that can be passed with the mount(2) system call. A versioning file system, for example, can be passed a number indicating the maximum number of versions to allow per file. FiST can also define new error codes that can be returned to user processes, for the latter to understand additional modes of failure. For example, an encryption file system can return a new error code indicating that the cipher key in use has expired.

FiST Rules define actions that generally determine the behavior for individual files. 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 target file, so it might be restored later. We separated the declarations and rules sections for programming ease: developers know that global declarations go in the former, and actions that affect vnode operations go in the latter.

Additional C Code includes additional C functions that might be referenced by code in the rest of the file system. 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. This section provides a flexible extension mechanism for FiST-based file systems. Code in this section may use any basic FiST primitives (discussed in Section 2.3.1) which are helpful in writing portable code. We also allow developers to write code that takes advantage of system-specific features; this flexibility, however, may result in non-portable code.

The remainder of this section introduces the FiST language primitives, the various participants in a file system (such as files, mounts, and processes), their attributes and how to extend them and store them persistently, and how to control the execution flow in a file system. The examples in Section 4 are also helpful because they further illustrate the FiST language.

2.3.1 FiST Syntax

FiST syntax allows referencing mounted file systems and files, accessing attributes, and calling FiST functions. Mount references begin with `$vfs`, while file references use a shorter "$" syntax because we expect them to appear more often in FiST code. References may be followed by a name or number that distinguishes among multiple instances (e.g., `$1`, `$2`, etc.) especially useful when fan-out is used (Figure 4). Attributes of mounts and files are specified by appending a dot and the attribute name to the reference (e.g., `$vfs.blocksize`, `$1.name`, `$2.owner`, etc.) The scope of these references is the current vnode function in which they are executing.

There is only one instance of a running operating system. Similarly, there is only one process context executing that the file system has to be concerned with. Therefore FiST need only refer to their attributes. These read-only attributes are summarized in Table 2. The scope of all read-only "%" attributes is global.

FiST code can call FiST functions from anywhere in the file system, some of which are shown in Table 3. The scope of FiST functions is global in the mounted file system. These functions form a comprehensive library of portable

Global	Meaning
%blocksize	native disk block size
%gid	effective group ID
%pagesize	native page size
%pid	process ID
%time	current time (seconds since epoch)
%uid	effective user ID

Table 2: Global Read-Only FiST Variables

routines useful in writing file systems. The names of these functions begin with "fist." FiST functions can take a variable number of arguments, omit some arguments where suitable defaults exist, and use different types for each argument. These are true functions that can be nested and may return any single value.

Function	Meaning
fistPrintf	print messages
fistStrEq	string comparison
fistMemCpy	buffer copying similar
fistLastErr	get the last error code
fistSetErr	set the return error code
fistReturnErr	return an error code immediately
fistSetIoctlData	set ioctl value to pass to a user process
fistGetIoctlData	get ioctl value from a user process
fistSetFileData	write arbitrary data to a file
fistGetFileData	read arbitrary data from a file
fistLookup	find a file in a directory
fistReaddir	read a directory
fistSkipName	hide a name of a file in a directory
fistOp	execute an arbitrary vnode operation

Table 3: A sample of FiST functions used in this paper

Each mount and file has attributes associated with it. FiST recognizes common attributes of mounted file systems and files that are defined by the system, such as the name, owner, last modification time, or protection modes. FiST also allows developers to define new attributes and optionally store them persistently. Attributes are accessed by appending the name of the attribute to the mount or file reference, with a single dot in between, much the same way that C dereferences structure field names. For example, the native block size of a mounted file system is accessed as `$vfs.blocksize` and the name of a file is `$0.name`.

FiST allows users to create new file attributes. For example, an ACL file system may wish to add timed access to certain files. The following FiST Declaration can define the new file attributes in such a file system:

```
per_vnode {
    int     user;    /* extra user */
    int     group;   /* extra group */
    time_t  expire;  /* access expiration time */
};
```

With the above definition in place, a FiST file system may refer to the additional user and group who are allowed to access the file as `$0.user` and `$0.group`, respectively. The expiration time is accessed as `$0.expire`.

The `per_vnode` declaration defines new attributes for files, but those attributes are only kept in memory. FiST also provides different methods to define, store, and access additional attributes persistently. This way, a file system developer has the flexibility of deciding if new attributes need only remain in memory or saved more permanently.

For example, an encrypting file system may want to store an encryption key, cipher ID, and Initialization Vector (IV) for each file. This can be declared in FiST using:

```
fileformat SECDAT {
    char  key[16];    /* cipher key */
    int   cipher;     /* cipher ID */
    char  iv[16];     /* initialization vector */
};
```

Two FiST functions exist for handling file formats: fistSetFileData and fistGetFileData. These two routines can store persistently and retrieve (respectively) additional file system and file attributes, as well as any other arbitrary data. For example, to save the cipher ID in a file called `.key`, use:

```
int cid;
/* set cipher ID */
fistSetFileData(".key", SECDAT, cipher, cid);
```

The above FiST function will produce kernel code to open the file named "`.key`" and write the value of the "cid" variable into the "cipher" field of the "SECDAT" file format, as if the latter had been a data structure stored in the "`.key`" file.

Finally, the mechanism for adding new attributes to mounts is similar. For files, the declaration is `per_vnode` while for mounts it is `per_vfs`. The routines fistSetFileData and fistGetFileData can be used to access any arbitrary persistent data, for both mounts and files.

2.3.2 Rules for Controlling Execution and Information Flow

In the previous sections we considered how FiST can control the flow of information between the various layers. In this section we describe how FiST can control the execution flow of various operations using FiST rules.

FiST does not change the interfaces that call it, because such changes will not be portable across operating systems and may require changing many user applications. FiST therefore only exchanges information with applications using existing APIs (e.g., ioctls) and those specific applications can then affect change.

The most control FiST file systems have is over the file system (vnode) operations that execute in a normal stackable setting. Figure 6 highlights what a typical stackable vnode operation does: (1) find the vnode of the lower level mount, and (2) repeat the same operation on the lower vnode.

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

  /* (1) pre-call code goes here */
  /* (2) call same operation on lower file system */
  error = VOP_GETATTR(lower_vp, args...);
  /* (3) post-call code goes here */
  return error;
}
```

Figure 6: A skeleton of typical kernel C code for stackable vnode functions. FiST can control all three sections of every vnode function: pre-call, post-call, and the call itself.

The 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 mount. Next, the function actually calls the lower level mounted 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 mount passed to the function.

There are three key parts in any stackable function that FiST can control: the code that may run before calling the lower level mount (pre-call), the code that may run afterwards (post-call), and the actual call to the lower level mount. FiST can insert arbitrary code in the pre-call and post-call sections, as well as replace the call part itself with anything else.

By default, the pre-call and post-call sections are empty, and the call section contains code to pass the operation to the lower level file system. These defaults produce a file system that stacks on another but does not change behavior, and that was designed so developers do not have to worry about the basic stacking behavior—only about their changes.

For example, a useful pre-call code in an encryption file system would be to verify the validity of cipher keys. A replication file system may insert post-call code to repeat the same vnode operation on other replicas. A versioning file system could replace the actual call to remove a file with a call to rename it; an example FiST code for the latter might be:

```
%op:unlink:call {
  fistRename($name, fistStrAdd($name, ".unrm"));
}
```

The general form for a FiST rule is:

$$\%callset : optype : part \; \{code\} \qquad (1)$$

Table 4 summarizes the possible values that a FiST rule can have. *Callset* defines a collection of operations to operate on. *Optype* further defines the call set to a subset of operations or a single operation. *Part* defines the part of the call that the following code refers to: pre-call, call, post-call, or the name of a newly defined ioctl. Finally, *code* contains any C code enclosed in braces.

Call Sets	
op	to refer to a single operation
ops	to refer to all operations
readops	to refer to non state changing operations
writeops	to refer to state changing operations
Operation Types	
all	all operations
data	operations that manipulate file data
name	operations that manipulate file names
The rest of the operation types specify one of the following vnode operations: create, getattr, l/stat, link, lookup, mkdir, read, readdir, readlink, rename, rmdir, setattr, statfs, symlink, unlink, and write.	
Call Part	
precall	part before calling the lower file system
call	the actual call to the lower file system
postcall	part after calling the lower file system
ioctl	name of a newly defined ioctl

Table 4: Possible Values in FiST Rules

2.3.3 Filter Declarations and Filter Functions

FiST file systems can perform arbitrary manipulations of the data they exchange between layers. The most useful and at the same time most complex data manipulations in a stackable file system involve file data and file names. To manipulate them consistently without FiST or Wrapfs, 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, mkdir, 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 declare `filter:data` in their FiST file, fistgen looks for two data coding functions in the Additional C Code section of the FiST File: `encode_data` and `decode_data`. These functions take an input data page, and an allocated output page of the same size. 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 output page by encoding or decoding it appropriately and return a success or failure status code. Our encryption file system uses a data filter to encrypt and decrypt data (Section 4.1).

With the FiST declaration `filter:name`, fistgen inserts code and calls to encode or decode strings representing file names. The file name coding functions (`encode_name` and `decode_name`) take an input file name string and its length. They must allocate a new string 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.

Using FiST filters, developers can easily produce file systems that perform complex manipulations of data or names exchanged between file system layers.

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. In particular, fistgen conditionally includes large portions of code that support FiST filters: code to manipulate file data or file names. 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 code generated by fistgen may contain automatically generated functions that are necessary to support proper FiST function semantics. 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 even edit that code and add more features by hand, if they so choose.

Fistgen also produces real C functions for specialized FiST syntax that cannot be trivially handled in C. 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 functions only if needed and only once.

2.5 Basefs

Basefs is a template system which was derived from Wrapfs[27]. It provides basic stacking functionality without changing other file systems or the kernel. To achieve this functionality, the kernel must support three features. First, in each of the VFS data structures, Basefs requires a field to store pointers to data structures at the layer below. Second, new file systems should be able to call VFS functions. Third, the kernel should export all symbols that may be needed by new loadable kernel modules. The last two requirements are needed only for loadable kernel modules.

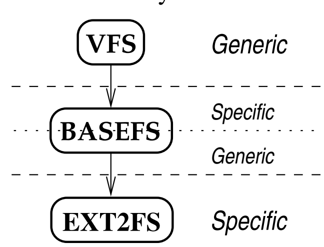

Figure 7: Where Basefs fits inside the kernel

Basefs handles many of the internal details of operating systems, thus freeing developers from dealing with kernel specifics. Basefs provides a stacking layer that is independent from the layers above and below it. Figure 7 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.

Basefs performs all data reading and writing on whole pages. This simplifies mixing regular reads and writes with memory-mapped operations, and gives developers a single paged-based interface to work with. Currently, file systems derived from Basefs manipulate data in whole pages and may not change the data size (e.g., compression).

To improve performance, Basefs copies and caches data pages in its layer and the layers below it.[1] Basefs saves memory by caching at the lower layer only if file data is manipulated and fan-in was used; these are the usual conditions that require caching at each layer.

Basefs is different from Wrapfs in four ways. First, substantial portions of code to manipulate file data and file names, as well as debugging code are not included in Basefs by default. 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 exceeding the performance of other layered file systems was one of the design goals for Basefs.

Second, Basefs adds support for fan-out file systems natively. This code is also conditionally included, because it is more complex than single-stack file systems, adds more performance overhead, and consumes more memory. A complete discussion of the implementation and behavior of fan-out file systems is beyond the scope of this paper.

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 a library of common functions: opening, reading or writing, and then closing arbitrary files; storing extended attributes persistently; user-level utilities to mount and unmount file systems, as well as manipulate ioctls; inspecting and modifying file attributes, and more.

Fourth, 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.

The implementation of Basefs was simple and improved on previously reported efforts[27]. No changes were required to either Solaris or FreeBSD. No changes to Linux were required if using statically linked modules. To use dynamically loadable kernel modules under Linux, only three lines of code were changed in a header file. This change was passive and did not have any impact on the Linux kernel.

The remainder of this section describes the implementation of fistgen. Fistgen translates FiST code into C code which implements the file system described in the FiST input file. The code can be compiled as a dynamically loadable kernel module or statically linked with a kernel. In this section we describe the implementation of key features of FiST that span its full range of capabilities.

We implemented read-only execution environment variables (Section 2.3.1) 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 `VOP_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, `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 can access it as any other vnode attribute, or any other per-vnode attribute added using the `per_vnode` declaration. We implemented all other fields defined using the `per_vfs` FiST declaration in a similar fashion.

The FiST declarations described in Section 2.3 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 fan-in mount style by ex-

[1] Heidemann proposed a solution to the cache coherency problem through a centralized cache manager[6]. His solution, however, required modifications to existing file systems and the rest of the kernel.

cluding 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 functions, fistGetIoctlData and fistSetIoctlData (Section 2.2), 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 functions used with it: fistGetFileData and fistSetFileData (Section 2.3.1). Consider this small code excerpt:

```
fileformat 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. To improve performance, if fileformat related functions are called several times inside a vnode function, we keep the file they refer to open until the last call that uses it.

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).

4 Examples

This section describes the design and implementation of several 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. **Cryptfs**: is an encryption file system.
2. **Aclfs**: adds simple access control lists.
3. **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 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. An additional example, Snoopfs, was described in Section 2.1.

4.1 Cryptfs

Cryptfs is a strong encryption file system. It uses the Blowfish[21] encryption algorithm in Cipher Feedback (CFB) mode[20]. 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[26].

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. Some of the more important 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;
}
...
```

The above example omits 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 4.2.); using several types of encryption algorithms, and defining mount flags to select among them.

4.2 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: disallowing fan-in (more secure), using special purpose auxiliary files, and hiding files from users' view. The FiST code for Aclfs uses the FiST Declarations and FiST Rules sections:

```
fanin no;
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.

4.3 Unionfs

Unionfs joins the contents of two file systems similar to the union mounts in BSD-4.4[15] and Plan 9[17]. 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 code size is reduced dramatically when using FiST, and the corresponding improvements in development and porting times. 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 this paper: Snoopfs, Cryptfs, Aclfs, and Unionfs. These were tested 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 stand-alone 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 (modulo white-spaces and comments) to the hand-written code. We chose to include results for both Basefs and Wrapfs because the latter was released last year, and includes code that makes writing some file systems easier with Wrapfs than Basefs directly.

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. We then averaged the code sizes for the three platforms we implemented the file systems on: Linux 2.3, Solaris 2.6, and FreeBSD 3.3. These results are shown in Figure 8. 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.

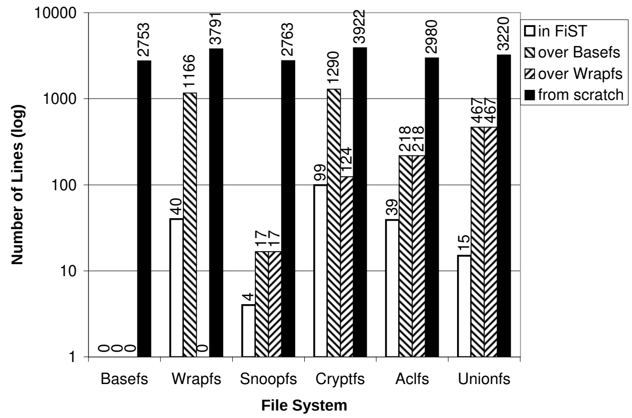

Figure 8: Average code size for various file systems when written in FiST, written given the Basefs or Wrapfs templates, and written from scratch in C.

Figure 8 shows large reductions in code size when comparing FiST versus code hand-written from scratch—generally writing tens of lines instead of thousands. We also include results for the two templates. Size reductions for the four example file systems range from a factor of 40 to 691, with an average of 255. We focus though on the comparison of FiST versus stackable template systems. As Wrapfs represents the most conservative comparison, the figure shows for each file system the additional number of lines of code written using Wrapfs. The smallest average code size reduction in using FiST versus Wrapfs or Basefs across all four file systems ranges from a factor of 1.3 to 31.1; the average reduction rate is 10.5.

Figure 8 suggests two size reduction classes. First, moderate (5–6 times) savings are achieved for Snoopfs, Cryptfs, and Aclfs. The reason for this is that some lines of FiST code for these file systems produce ten or more lines of C code, while others result in almost a one-to-one translation in terms of number of lines.

Second, 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 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. 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.[2]

Figure 8 shows the code sizes for *each* platform. The savings gained by FiST are multiplied 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 when comparing FiST to code written using the templates. This aggregated reduction factor exceeds 750 times when comparing FiST to C code written from scratch. The more ports of Basefs 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; these figures do not represent a controlled study. Figure 9 shows the number of days we spent developing various file systems and porting them to three different platforms.

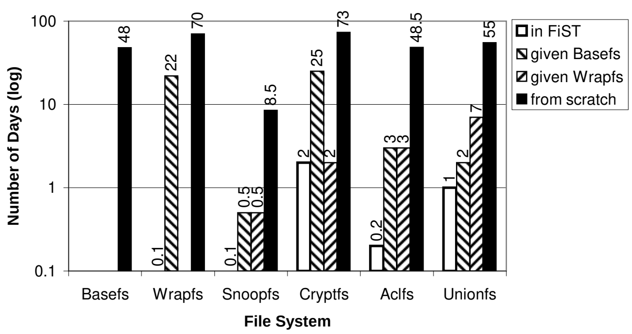

Figure 9: Average estimated reduction in development time

We estimated the incremental time spent designing, developing, and debugging 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.

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 rest of the support for Cryptfs is already in Wrapfs.

The average per platform reduction in development time across the four file systems is a factor of seven in using FiST versus the Wrapfs templates. If we assume that development time correlates directly to productivity, we can corroborate our results with Brooks's report that high-level languages are responsible for at least a factor of five in improved productivity[3].

An additional metric of productivity is comparing the number of lines of C code developed for each man-day, given the templates. The average number of lines of code we wrote per man-day was 80. One user of our Wrapfs templates had used them to create a new migration file system called mfs[3]. The average number of lines of code he wrote per man-day was 68. The difference between his rate of productivity and ours is only 20%, which can be explained because we are more experienced in writing file systems than he is.

The most obvious savings in development time come when taking into account multiple platforms. Then it is clearer that each additional platform increases the savings factor of FiST versus other methods by one more.

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 by building a large package: am-utils-6.0, which contains about 50,000 lines of C code in several dozen small files and builds eight binaries; the build process contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations. Each benchmark was run once to warm up the cache for executables, libraries, and header files which are *outside* the tested file system; this result was discarded. Afterwards, we took 10 new measurements and averaged them. In between each test, we unmounted the tested file system and the one below it, and then remounted them; this ensured that we started each test on a cold cache for that file system. The standard deviations for our measurements were 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.

Figure 10 shows the performance overhead of each file system compared to the one it was based on. The intent of these figures is two-fold: (1) to show that the basic stacking overhead is small, and (2) to show the performance benefits

[2]Unfortunately, the stacking infrastructure in FreeBSD is currently broken, so we were unable to compare the performance of our stacking to FreeBSD's.

[3]http://www-internal.alphanet.ch/~schaefer/mfs.html

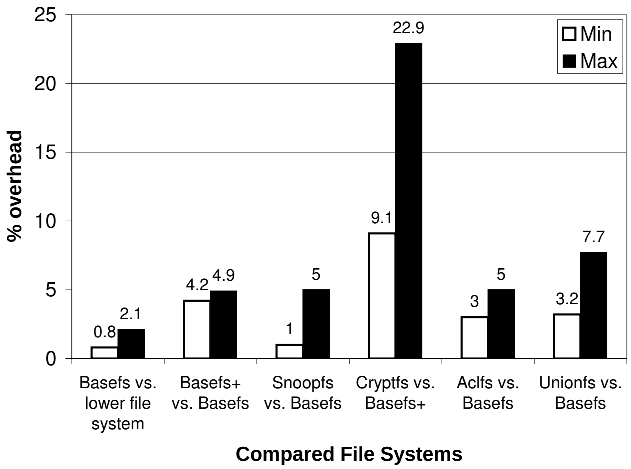

Figure 10: Performance overhead of various file systems for the large compile benchmark, across three operating systems

of conditionally including code for manipulating file names and file data in Basefs. Basefs+ refers to Basefs with code for manipulating file names and file data.

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[8, 22]. In addition, the overhead of the example file systems due to new file system functionality is greater than the basic stacking overhead imposed by our templates in all cases, even for very simple file systems. With regard to performance, developers who extend file system functionality using FiST primarily need to be concerned with the performance cost of new file system functionality as opposed to the cost of the FiST stacking infrastructure. For instance, the overhead of Cryptfs is the largest of all the file systems shown due to the cost of the Blowfish cipher. Note that the performance of individual file systems can vary greatly depending on the operating system in question.

Figure 10 also shows the benefits of having FiST customize the generated file system infrastructure based on the file system functionality required. The comparison of Basefs+ versus Basefs shows that the overhead of including code for manipulating file names and file data is 4.2–4.9% over Basefs. This added overhead is not incurred in Basefs unless the file systems derived from it requires file data or file name manipulations. While Cryptfs requires Basefs+ functionality, Snoopfs, Aclfs, and Unionfs do not. Compared to a stackable file system such as Wrapfs, FiST's ability to conditionally include file system infrastructure code saves an additional 4% of performance overhead for Snoopfs, Aclfs, and Unionfs.

We also performed several micro-benchmarks which included a series of recursive copies (cp –r), recursive removals (rm –rf), recursive find, and "find-grep" (find /mnt –print | xargs grep *pattern*) using the same file set used for the large compile. The focus of this paper is not on performance, but on savings in code size and development time. Since the micro-benchmarks confirmed our previous good results, we do not repeat them here[27].

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[19]. A few other projects followed his, including further prototypes for extensible file systems in SunOS[22], and the Ficus layered file system[5, 7]. Webber implemented file system interface extensions that allow user-level file servers[25]. 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 operating systems, such as the HURD[4], Spring[13], and the Exokernel[10] 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[13]. 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[12]. 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[10]. 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.

High level languages have seldom been used to generate code for operating system components. FiST is the first major language to describe a large component of the operating system, the file system. Previous work in the area of operating system component languages includes a language to describe video device drivers[24].

7 Conclusions

The main contribution of this work is the FiST language which can describe stackable file systems. This is a first time a high-level language has been used to 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 about seven times faster than when we wrote the code using Basefs. 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 multiplied 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

We are developing support for file systems that change sizes such as for compression. The main complexity with supporting compression is that the file offsets at the upper and lower layers are no longer identical, and some form of efficient mapping is needed for operations such as appending to a file or writing in the middle. This code complicates the templates, but makes no change to the language.

We are also 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.

We 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[14]. 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. We estimate that porting Basefs to NT will take 2–3 months, not 1–3 weeks as we predict for Unix ports.

8 Acknowledgments

We would like to thank the anonymous USENIX reviewers and our shepherd Keith Smith, for their helpful comments in reviewing this paper. This work was partially made possible by NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374.

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, Summer 1986.

[2] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. *USENIX Conf. Proc.*, pages 267–75, Winter 1988.

[3] F. Brooks. "No Silver Bullet" Refired. In *The Mythical Man-Month, Anniversary Ed.*, pages 205–26. Addison-Wesley, 1995.

[4] 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`.

[5] 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, Summer 1990.

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

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

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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, Winter 1995.

[16] J. S. Pendry and N. Williams. Amd – The 4.4 BSD Automounter. User Manual, edition 5.3 alpha. March 1991.

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

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

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

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

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

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

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

[24] S. Thibault, R. Marlet, and C. Consel. A Domain Specific Language for Video Device Drivers: From Design to Implementation. *USENIX Conf. on Domain-Specific Languages*, pages 11–26, 1997.

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

[26] 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.

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

Software, documentation, and additional papers are available from http://www.cs.columbia.edu/~ezk/research/fist.