Extending File Systems Using Stackable Templates

Erez Zadok, Ion Badulescu, and Alex Shender
Computer Science Department, Columbia University
{ezk,ion,alex}@cs.columbia.edu

Abstract

Extending file system functionality is not a new idea, but a desirable one nonetheless[6, 14, 18]. In the several years since stackable file systems were first proposed, only a handful are in use[12, 19]. Impediments to writing new file systems include the complexity of operating systems, the difficulty of writing kernel-based code, the lack of a true stackable vnode interface[14], and the challenges of porting one file system to another operating system.

We advocate writing new stackable file systems as kernel modules. As a starting point, we propose a portable, stackable template file system we call Wrapfs (wrapper file system). Wrapfs is a canonical, minimal stackable file system that can be used as a pattern across a wide range of operating systems and file systems. Given Wrapfs, developers can add or modify only that which is necessary to achieve the desired functionality. Wrapfs takes care of the rest, and frees developers from the details of operating systems. Wrapfs templates exist for several common operating systems (Solaris, Linux, and FreeBSD), thus alleviating portability concerns. Wrapfs can be ported to any operating system with a vnode interface that provides a private data pointer for each data structure used in the interface. The overhead imposed by Wrapfs is only 5–7%.

This paper describes the design and implementation of Wrapfs, explores portability issues, and shows how the implementation was achieved without changing client file systems or operating systems. We discuss several examples of file systems written using Wrapfs.

1 Introduction

Adding functionality to existing file systems in an easy manner has always been desirable. Several ideas have been proposed and some prototypes implemented[6, 14, 18]. None of the proposals for a new extensible file system interface has made it to commonly used Unix operating systems. The main reasons are the significant changes that overhauling the file system interface would require, and the impact it would have on performance.

Kernel-resident native file systems are those that interact directly with lower level media such as disks[9] and networks[11, 16]. Writing such file systems is difficult because it requires deep understanding of specific operating system internals, especially the interaction with device drivers and the virtual memory system. Once such a file system is written, porting it to another operating system is just as difficult as the initial implementation, because specifics of different operating systems vary significantly.

Others have resorted to writing file systems at the user level. These file systems work similarly to the Amd automounter[13] and are based on an NFS server. While it is easier to develop user-level file servers, they suffer from poor performance due to the high number of context switches they incur. This limits the usefulness of such file systems. Later works, such as Autofs[3], attempt to solve this problem by moving critical parts of the automounter into the kernel.

We propose a compromise solution to these problems: writing kernel resident file systems that use existing native file systems, exposing to the user a vnode interface that is similar even across different operating systems. Doing so results in performance similar to that of kernel-resident systems, with development effort on par with user level file systems. Specifically, we provide a template *Wrapper File System* called Wrapfs. Wrapfs can *wrap* (mount) itself on top of one or more existing directories, and act as an intermediary between the user accessing the mount point and the lower level file system it is mounted on. Wrapfs can transparently change the behavior of the file system, while keeping the underlying media unaware of the upper-level changes. The Wrapfs template takes care of many file system internals and operating system bookkeeping, and it provides the developer with simple hooks to manipulate the data and attributes of the lower file system's objects.

1.1 The Stackable Vnode Interface

Wrapfs is implemented as a stackable vnode interface. A *Virtual Node* or *vnode* is a data structure used within Unix-based operating systems to represent an open file, directory, or other entities that can appear in the file system namespace. A vnode does not expose what type of physical file system it implements. The *vnode interface* allows higher level operating system modules to perform operations on vnodes uniformly. The *virtual file system* (VFS) contains the common file system code of the vnode interface.

One improvement to the vnode concept is *vnode stacking*[6, 14, 18], a technique for modularizing file system functions by allowing one vnode interface implementation to call another. Before stacking existed, there was only one vnode interface implementation; higher level operating system code called the vnode interface which in turn called code for a specific file system. With vnode stacking, several implementations may exist and may call each other in sequence: the code for a certain operation at stack level N typically calls the corresponding operation at level $N - 1$, and so on.

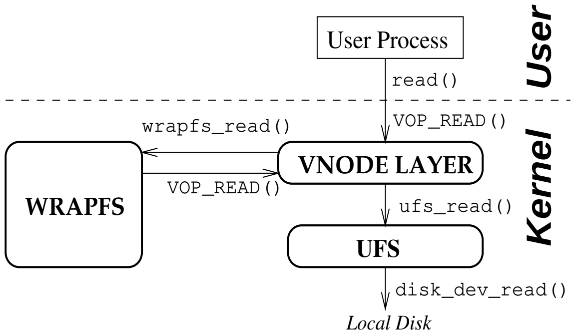

Figure 1: A Vnode Stackable File System

Figure 1 shows the structure for a simple, single-level stackable wrapper file system. System calls are translated into VFS calls, which in turn invoke their Wrapfs equivalents. Wrapfs again invokes generic VFS operations, and the latter call their respective *lower level* file system operations. Wrapfs calls the lower level file system without knowing who or what type it is.

The rest of this paper is organized as follows. Section 2 discusses the design of Wrapfs. Section 3 details Wrapfs's implementation, and issues relating to its portability to various platforms. Section 4 describes four example file systems written using Wrapfs; Section 5 evaluates their performance and portability. We survey related works in Section 6 and conclude in Section 7.

2 Design

Our main design considerations for Wrapfs were:

1. What developers want to change in a file system.
2. What API should Wrapfs offer these users that would free them from operating system internals.
3. How to allow advanced users the flexibility to control and manipulate all aspects of the file system.
4. Interaction of caches among different layers.
5. What user level mounting-related issues are there.
6. Performance overhead of Wrapfs.

The first five points are discussed below. Performance is addressed in detail in Section 5.

2.1 What to Change in a File System

As shown in Figure 1, Wrapfs is independent of the host system's vnode interface. Since most UNIX-like systems (including those that currently support Wrapfs) have static vnode interfaces, this means that Wrapfs cannot introduce fundamentally new vnode operations.[1] (Limited new functionality can be added using new `ioctl(2)` calls.) Our stackable file system architecture can, however, manipulate data, names, and attributes of files. We let Wrapfs users manipulate any of these.

The most obvious manipulation Wrapfs users want to do is of file data; that is useful for many applications, for example in encryption file systems. The next most likely item to manipulate is the file name. For example, an encryption file system can encrypt file names as well as data. A file system that translates between Unix and MS-DOS style file names can uniquely map long mixed-case Unix file names to 8.3-format upper-case names.

Finally, there are file attributes that users might want to change. For example, a file system can ensure that all files within it are world readable, regardless of the individual umask used by the users creating them. This is useful in directories where all files must be world-readable (such as html files served by most Web servers). Another file system might prevent anyone from setting the uid bit on executables, as a simple form of intrusion avoidance.

The aforementioned list of examples is not exhaustive, but only a hint of what can be accomplished with level of flexibility that Wrapfs offers. Wrapfs's developer API is summarized in Table 1 and is described below.

2.1.1 File Data API

The system call interface offers two methods for reading and writing file data. The first is by using the `read` and `write` system calls. The second is via the MMAP interface. The former allows users to manipulate arbitrary amounts of file data. In contrast, the MMAP interface operates on a file in units of the native page size. To accommodate the MMAP interface, we decided to require file system developers using Wrapfs to also manipulate file data on whole pages. Another reason for manipulating only whole pages was that some file data changes may require it. Some encryption algorithms work on fixed size data blocks and bytes within the block depend on preceding bytes.

[1] The UCLA stackable file system replaced the static UNIX vnode interface with a dynamic interface that allowed file system developers to introduce new operations[6].

Call	Input Argument	Output Argument
encode_data	buffer from user space	encoded (same size) buffer to be written
decode_data	buffer read from lower level file system	decoded (same size) buffer to pass to user space
encode_filename	file name passed from user system call	encoded (and allocated) file name of any length to use in lower level file system
decode_filename	file name read from the lower level file system	decoded (and allocated) file name of any length to pass back to a user process
other	Inspect or modify file attributes in vnode functions, right before or after calling lower level file system	

Table 1: Wrapfs Developer API

All vnode calls that write file data call a function `encode_data` before writing the data to the lower level file system. Similarly, all vnode calls that read file data call a function `decode_data` after reading the data from the lower level file system. These two functions take two buffers of the same size: one as input, and another as output. The size of the buffer can be defined by the Wrapfs developer, but it must be an even multiple of the system's page size, to simplify handling of MMAP functions. Wrapfs passes other auxiliary data to the encode and decode functions, including the file's attributes and the user's credentials. These are useful when determining the proper action to take. The encode and decode functions return the number of bytes manipulated, or a negative error code.

All vnode functions that manipulate file data, including the MMAP ones, call either the encode or decode functions at the right place. Wrapfs developers who want to modify file data need not worry about the interaction between the MMAP, read, and write functions, about file or page locks, reference counts, caches, status flags, and other bookkeeping details; developers need only to fill in the encode and decode functions appropriately.

2.1.2 File Names API

Wrapfs provides two file name manipulating functions: `encode_filename` and `decode_filename`. They take in a single file name component, and ask the Wrapfs developer to fill in a new encoded or decoded file name of any length. The two functions return the number of bytes in the newly created string, or a negative error code. Wrapfs also passes to these functions the file's vnode and the user's credentials, allowing the function to use them to determine how to encode or decode the file name. Wrapfs imposes only one restriction on these file name manipulating functions. They must not return new file names that contain characters illegal in Unix file names, such as a null or a "/".

The user of Wrapfs who wishes to manipulate file names need not worry about which vnode functions use file names, or how directory reading (`readdir`) is being accomplished. The file system developer need only fill in the file name encoding and decoding functions. Wrapfs takes care of all other operating system internals.

2.1.3 File Attributes

For the first prototype of Wrapfs, we decided not to force a specific API call for accessing or modifying file attributes. There are only one or two places in Wrapfs where attributes are handled, but these places are called often (i.e., lookup). We felt that forcing an API call might hurt performance too much. Instead, we let developers inspect or modify file attributes directly in Wrapfs's source.

2.2 User Level Issues

There are two important issues relating to the extension of the Wrapfs API to user-level: mount points and ioctl calls.

Wrapfs can be mounted as a regular mount or an overlay mount. The choice of mount style is left to the Wrapfs developer. Figure 2 shows an original file system and the two types of mounts, and draws the boundary between Wrapfs and the original file system after a successful mount.

In a regular mount, Wrapfs receives two pathnames: one for the mount point (`/mnt`), and one for the directory to stack on (the mounted directory `/usr`). After executing, for example, `mount -t wrapfs /mnt /usr`, there are two ways to access the mounted-on file system. Access via the mounted-on directory (`/usr/ucb`) yields the lower level files without going through Wrapfs. Access via the mount point (`/mnt/ucb`), however, goes through Wrapfs first. This mount style exposes the mounted directory to user processes; it is useful for debugging purposes and for applications (e.g., backups) that do not need the functionality Wrapfs implements. For example, in an encryption file system, a backup utility can backup files faster and safer if it uses the lower file system's files (ciphertext), rather than the ones through the mount point (cleartext).

In an overlay mount, accomplished using `mount -t wrapfs -O /usr`, Wrapfs is mounted directly on top of `/usr`. Access to files such as `/usr/ucb` go though Wrapfs. There is no way to get to the original file system's files under `/usr` without passing through Wrapfs first. This mount style has the advantage of hiding the lower level file system from user processes, but may make backups and debugging harder.

The second important user-level issue relates to the `ioctl(2)` system call. Ioctls are often used to extend file

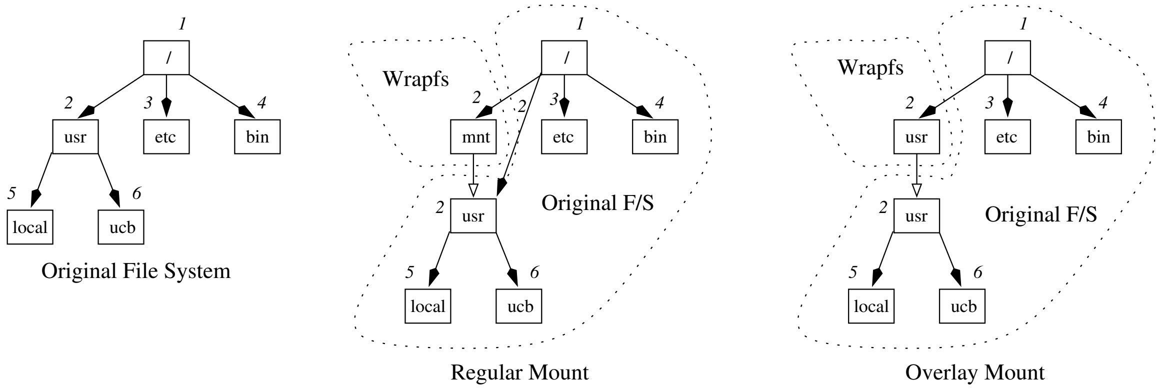

Figure 2: Wrapfs Mount Styles

system functionality. Wrapfs allows its user to define new ioctl codes and implement their associated actions. Two ioctls are already defined: one to set a debugging level and one to query it. Wrapfs comes with lots of debugging traces that can be turned on or off at run time by a root user. File systems can implement other ioctls. An encryption file system, for example, can use an ioctl to set encryption keys.

2.3 Interaction Between Caches

When Wrapfs is used on top of a disk-based file system, both layers cache their pages. Cache incoherency could result if pages at different layers are modified independently. A mechanism for keeping caches synchronized through a centralized cache manager was proposed by Heidemann[5]. Unfortunately, that solution involved modifying the rest of the operating system and other file systems.

Wrapfs performs its own caching, and does not explicitly touch the caches of lower layers. This keeps Wrapfs simpler and more independent of other file systems. Also, since pages can be served off of their respective layers, performance is improved. We decided that the higher a layer is, the more authoritative it is: when writing to disk, cached pages for the same file in Wrapfs overwrite their UFS counterparts. This policy correlates with the most common case of cache access, through the uppermost layer. Finally, note that a user process can access cached pages of a lower level file system only if it was mounted as a regular mount (Figure 2). If Wrapfs is overlay mounted, user processes could not access lower level files directly, and cache incoherency for those files is less likely to occur.

3 Implementation

This section details the more difficult parts of the implementation and unexpected problems we encountered. Our first implementation concentrated on the Solaris 2.5.1 operating system because Solaris has a standard vnode interface and we had access to kernel sources. Our next two implementations were for the Linux 2.0 and the FreeBSD 3.0 operating systems. We chose these two because they are popular, are sufficiently different, and they also come with kernel sources. In addition, all three platforms support loadable kernel modules, which made debugging easier. Together, the platforms we chose cover a large portion of the Unix market.

The discussion in the rest of this section concentrates mostly on Solaris, unless otherwise noted. In Section 3.5 we discuss the differences in implementation between Linux and Solaris. Section 3.6 discusses the differences for the FreeBSD port.

3.1 Stacking

Wrapfs was initially similar to the Solaris loopback file system (lofs)[19]. Lofs passes all Vnode/VFS operations to the lower layer, but it only stacks on directory vnodes. Wrapfs stacks on every vnode, and makes identical copies of data blocks, pages, and file names in its own layer, so they can be changed independently of the lower level file system. Wrapfs does not explicitly manipulate objects in other layers. It appears to the upper VFS as a lower-level file system; concurrently, Wrapfs appears to lower-level file systems as an upper-layer. This allows us to stack multiple instances of Wrapfs on top of each other.

The key point that enables stacking is that each of the major data structures used in the file system (`struct vnode` and `struct vfs`) contain a field into which we can store file system specific data. Wrapfs uses that private field to store several pieces of information, especially a pointer to the corresponding lower level file system's vnode and VFS. When a vnode operation in Wrapfs is called, it finds the lower level's vnode from the current vnode, and repeats the same operation on the lower level vnode.

3.2 Paged Reading and Writing

We perform reading and writing on whole blocks of size matching the native page size. Whenever a read for a range of bytes is requested, we compute the extended range of bytes up to the next page boundary, and apply the operation to the lower file system using the extended range. Upon successful completion, the exact number of bytes requested are returned to the caller of the vnode operation.

Writing a range of bytes is more complicated than reading. Within one page, bytes may depend on previous bytes (e.g., encryption), so we have to read and decode parts of pages before writing other parts of them.

Throughout the rest of this section we will refer to the upper (wrapping) vnode as V, and to the lower (wrapped) vnode as V'; P and P' refer to memory mapped pages at these two levels, respectively. The example[2] depicted in Figure 3 shows what happens when a process asks to write bytes of an existing file from byte 9000 until byte 25000. Let us assume that the file in question has a total of 4 pages (32768) worth of bytes in it.

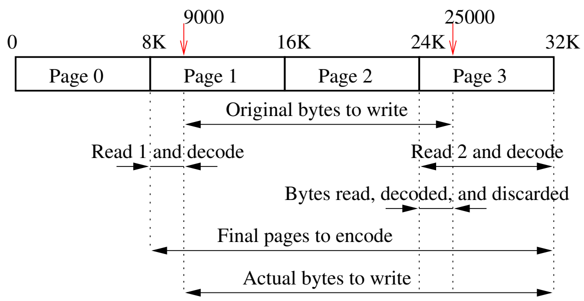

Figure 3: Writing Bytes in Wrapfs

1. Compute the extended page boundary for the write range as 8192–32767 and allocate three empty pages. (Page 0 of V is untouched.)
2. Read bytes 8192–8999 (page 1) from V', decode them, and place them in the first allocated page. We do not need to read or decode the bytes from 9000 onwards in page 1 because they will be overwritten by the data we wish to write anyway.
3. Skip intermediate data pages that will be overwritten by the overall write operation (page 2).
4. Read bytes 24576–32767 (page 3) from V', decode them, and place them in the third allocated page. This time we read and decode the whole page because we need the last $32767-25000=7767$ bytes and these bytes depend on the first $8192-7767=426$ bytes of that page.
5. Copy the bytes that were passed to us into the appropriate offset in the three allocated pages.
6. Finally, we encode the three data pages and call the write operation on V' for the same starting offset (9000). This time we write the bytes all the way to the last byte of the last page processed (byte 32767), to ensure validity of the data past file offset 25000.

[2]The example is simplified because it does not take into account sparse files, and appending to files.

3.2.1 Appending to Files

When files are opened for appending only, the VFS does not provide the vnode `write` function the real size of the file and where writing begins. If the size of the file before an append is not an exact multiple of the page size, data corruption may occur, since we will not begin a new encoding sequence on a page boundary.

We solve this problem by detecting when a file is opened with an append flag on, turn off that flag before the open operation is passed on to V', and replace it with flags that indicate to V' that the file was opened for normal reading and writing. We save the initial flags of the opened file, so that other operations on V could tell that the file was originally opened for appending. Whenever we write bytes to a file that was opened in append-only mode, we first find its size, and add that to the file offsets of the write request. In essence we convert append requests to regular write requests starting at the end of the file.

3.3 File Names and Directory Reading

Readdir is implemented in the kernel as a restartable function. A user process calls the readdir C library call, which is translated into repeated calls to the `getdents(2)` system call, passing it a buffer of a given size. The kernel fills the buffer with as many directory entries as will fit in the caller's buffer. If the directory was not read completely, the kernel sets a special EOF flag to false. As long as the flag is false, the C library function calls `getdents(2)` again.

The important issue with respect to directory reading is how to continue reading the directory from the offset where the previous read finished. This is accomplished by recording the last position and ensuring that it is returned to us upon the next invocation. We implemented `readdir` as follows:

1. A readdir vnode operation is called on V for N bytes worth of directory data.
2. Call the same vnode operation on V' and read back N bytes.
3. Create a new temporary buffer of a size that is as large as N.
4. Loop over the bytes read from V', breaking them into individual records representing one directory entry at a time (`struct dirent`). For each such, we call `decode_filename` to find the original file name. We construct a new directory entry record containing the decoded file name and add the new record to the allocated temporary buffer.
5. We record the offset to read from on the next call to `readdir`; this is the position past the last file name we just read and decoded. This offset is stored in one of the fields of the `struct uio` (representing data movement between user and kernel space) that is returned to the caller. A new structure is passed to

us upon the next invocation of readdir with the offset field untouched. This is how we are able to restart the call from the correct offset.

6. The temporary buffer is returned to the caller of the vnode operation. If there is more data to read from V', then we set the EOF flag to false before returning from this function.

The caller of `readdir` asks to read at most N bytes. When we decode or encode file names, the result can be a longer or shorter file name. We ensure that we fill in the user buffer with no more `struct dirent` entries than could fit (but fewer is acceptable). Regardless of how many directory entries were read and processed, we set the file offset of the directory being read such that the next invocation of the `readdir` vnode operation will resume reading file names from exactly where it left off the last time.

3.4 Memory Mapping

To support MMAP operations and execute binaries we implemented memory-mapping vnode functions. As per Section 2.3, Wrapfs maintains its own cached decoded pages, while the lower file system keeps cached encoded pages.

When a page fault occurs, the kernel calls the vnode operation `getpage`. This function retrieves one or more pages from a file. For simplicity, we implemented it as repeatedly calling a function that retrieves a single page—`getapage`. We implemented `getapage` as follows:

1. Check if the page is cached; if so, return it.
2. If the page is not cached, create a new page P.
3. Find V' from V and call the `getpage` operation on V', making sure it would return only one page P'.
4. Copy the (encoded) data from P' to P.
5. Map P into kernel virtual memory and decode the bytes by calling `wrapfs_decode`.
6. Unmap P from kernel VM, insert it into V's cache, and return it.

The implementation of `putpage` was similar to `getpage`. In practice we also had to carefully handle two additional details, to avoid deadlocks and data corruption. First, pages contain several types of locks, and these locks must be held and released in the right order and at the right time. Secondly, the MMU keeps mode bits indicating status of pages in hardware, especially the referenced and modified bits. We had to update and synchronize the hardware version of these bits with their software version kept in the pages' flags. For a file system to have to know and handle all of these low-level details blurs the distinction between the file system and the VM system.

3.5 Linux

When we began the Solaris work we referred to the implementation of other file systems such as lofs. Linux 2.0 did not have one as part of standard distributions, but we were able to locate and use a prototype[3]. Also, the Linux Vnode/VFS interface contains a different set of functions and data structures than Solaris, but it operates similarly.

In Linux, much of the common file system code was extracted and moved to a generic (higher) level. Many generic file system functions exist that can be used by default if the file system does not define its own version. This leaves the file system developer to deal with only the core issues of the file system. For example, Solaris User I/O (`uio`) structures contain various fields that must be updated carefully and consistently. Linux simplifies data movement by passing I/O related vnode functions a simple allocated (`char *`) buffer and an integer describing how many bytes to process in the buffer passed.

Memory-mapped operations are also easier in Linux. The vnode interface in Solaris includes functions that must be able to manipulate one or more pages. In Linux, a file system handles one page at a time, leaving page clustering and multiple-page operations to the higher VFS.

Directory reading was simpler in Linux. In Solaris, we read a number of raw bytes from the lower level file system, and parse them into chunks of `sizeof(struct dirent)`, set the proper fields in this structure, and append the file name bytes to the end of the structure (out of band). In Linux, we provide the kernel with a callback function for iterating over directory entries. This function is called by higher level code and ask us to simply process one file name at a time.

There were only two caveats to the portability of the Linux code. First, Linux keeps a list of exported kernel symbols (in `kernel/ksyms.c`) available to loadable modules. To make Wrapfs a loadable module, we had to export additional symbols to the rest of the kernel, for functions mostly related to memory mapping. Second, most of the structures used in the file system (`inode`, `super_block`, and `file`) include a private field into which stacking specific data could be placed. We had to add a private field to only one structure that was missing it, the `vm_area_struct`, which represents custom per-process virtual memory manager page-fault handlers. Since Wrapfs is the first fully stackable file system for Linux, we feel that these changes are small and acceptable, given that more stackable file systems are likely to be developed.[4]

[3] http://www.kvack.org/~blah/lofs/

[4] We submitted our small changes and expect that they will be included in a future version of Linux.

3.6 FreeBSD

FreeBSD 3.0 is based on BSD-4.4Lite. We chose it as the third port because it represents another major section of Unix operating systems. FreeBSD's vnode interface is similar to Solaris's and the port was straightforward. FreeBSD's version of the loopback file system is called *nullfs*[12], a template for writing stackable file systems. Unfortunately, ever since the merging of the VM and Buffer Cache in FreeBSD 3.0, stackable file systems stopped working because of the inability of the VFS to correctly map data pages of stackable file systems to their on-disk locations. We worked around two deficiencies in nullfs. First, writing large files resulted in some data pages getting zero-filled on disk; this forced us to perform all writes synchronously. Second, memory mapping through nullfs paniced the kernel, so we implemented MMAP functions ourselves. We implemented `getpages` and `putpages` using `read` and `write`, respectively, because calling the lower-level's page functions resulted in a UFS pager error.

4 Examples

This section details the design and implementation of four sample file systems we wrote based on Wrapfs. The examples range from simple to complex:

1. **Snoopfs**: detects and warns of attempted access to users' files by other non-root users.
2. **Lb2fs**: is a read-only file system that trivially balances the load between two replicas of the same file system.
3. **Usenetfs**: breaks large flat article directories (often found in very active news spools) into deeper directory hierarchies, improving file access times.
4. **Cryptfs**: is an encryption file system.

These examples are experimental and intended to illustrate the kinds of file systems that can be written using Wrapfs. We do not consider them to be complete solutions. Whenever possible, we illustrate potential enhancements to our examples. We hope to convince readers of the flexibility and simplicity of writing new file systems using Wrapfs.

4.1 Snoopfs

Users' home directory files are often considered private and personal. Normally, these files are read by their owner or by the root user (e.g., during backups). Other sanctioned file access includes files shared via a common Unix group. Any other access attempt may be considered a break-in attempt. For example, a manager might want to know if a subordinate tried to `cd` to the manager's `~/private` directory; an instructor might wish to be informed when anyone tries to read files containing homework solutions.

The one place in a file system where files are initially searched is the vnode `lookup` routine. To detect access problems, we first perform the lookup on the lower file system, and then check the resulting status. If the status was one of the error codes "permission denied" or "file not found," we know that someone was trying to read a file they do not have access to, or they were trying to guess file names. If we detect one of these two error codes, we also check if the current process belongs to the super-user or the file's owner by inspecting user credentials. If it was a root user or the owner, we do nothing. Otherwise we print a warning using the in-kernel log facility. The warning contains the file name to which access was denied and the user ID of the process that tried to access it.

We completed the implementation of Snoopfs in less than one hour (on all three platforms). The total number of lines of C code added to Wrapfs was less than 10.

Snoopfs can serve as a prototype for a more elaborate intrusion detection file system. Such a file system can prohibit or limit the creation or execution of setuid/setgid programs; it can also disallow overwriting certain executables that rarely change (such as `/bin/login`) to prevent attackers from replacing them with trojans.

4.2 Lb2fs

Lb2fs is a trivial file system that multiplexes file access between two identical replicas of a file system, thus balancing the load between them. To avoid concurrency and consistency problems associated with writable replicas, Lb2fs is a read-only file system: vnode operations that can modify the state of the lower file system are disallowed. The implementation was simple; operations such as `write`, `mkdir`, `unlink`, and `symlink` just return the error code "read-only file system." We made a simplifying assumption that the two replicas provide service of identical quality, and that the two remote servers are always available, thus avoiding fail-over and reliability issues.

The one place where new vnodes are created is in the `lookup` function. It takes a directory vnode and a pathname and it returns a new vnode for the file represented by the pathname within that directory. Directory vnodes in Lb2fs store not one, but two vnodes of the lower level file systems—one for each replica; this facilitates load-balancing lookups in directories. Only non-directories stack on top of one vnode, the one randomly picked. Lb2fs's lookup was implemented as follows:

1. An operation `lookup` is called on directory vnode DV and file name X.
2. Get from DV the two lower vnodes DV'_1 and DV'_2.
3. Pick one of the two lower vnodes at random, and repeat the lookup operation on it using X.
4. If the lookup operation for X was successful, then check the resulting vnode. If the resulting vnode was not a directory vnode, store it in the private data of DV and return.

5. If the resulting vnode was a directory vnode, then repeat the lookup operation on the *other* lower vnode; store the two resulting directory vnodes (representing X on the two replicas) in the private data of DV.

The implications of this design and implementation are twofold. First, once a vnode is created, all file operations using it go to the file server that was randomly picked for it. A lookup followed by an open, read, and close of a file, will all use the same file server. In other words, the granularity of our load balancing is on a per-file basis.

Second, since lookups happen on directory vnodes, we keep the two lower directory vnodes, one per replica. This is so we can randomly pick one of them to lookup a file. This design implies that every open directory vnode is opened on both replicas, and only file vnodes are truly randomly picked and load-balanced. The overall number of lookups performed by Lb2fs is twice for directory vnodes and only once for file vnodes. Since the average number of files on a file system is much larger than the number of directories, and directory names and vnodes are cached by the VFS, we expect the performance impact of this design to be small.

In less than one day we designed, implemented, tested, and ported Lb2fs. Many possible extensions to Lb2fs exist. It can be extended to handle three or a variable number of replicas. Several additional load-balancing algorithms can be implemented: round-robin, LRU, the most responsive/available replica first, etc. A test for downed servers can be included so that the load-balancing algorithm can avoid using servers that recently returned an I/O error or timed out (fail-over). Servers that were down can be added once again to the available pool after another timeout period.

4.3 Usenetfs

One cause of high loads on news servers in recent years has been the need to process many articles in very large flat directories representing newsgroups such as *control.cancel* and *misc.jobs.offered*. Significant resources are spent on processing articles in these few newsgroups. Most Unix directories are organized as a linear unsorted sequence of entries. Large newsgroups can have hundreds of thousands of articles in one directory, resulting in delays processing any single article.

When the operating system wants to lookup an entry in a directory with N entries, it may have to search all N entries to find the file in question. Table 2 shows the frequency of all file system operations that use a pathname on our news spool over a period of 24 hours.

Operation	**Frequency**	**% Total**
Lookup	7068838	88.41
Unlink	432269	5.41
Create	345647	4.32
Readdir	38371	0.48
All other	110473	1.38
Total	7995598	100.00

Table 2: Frequency of File System Operations on a News Spool

It shows that the bulk of all operations are for looking up files, so these should run very fast regardless of the directory size. Operations that usually run synchronously (`unlink` and `create`) account for about 10% of news spool activity and should also perform well on large newsgroups.

Usenetfs is a file system that rearranges the directory structure from being flat to one with small directories containing fewer articles. By breaking the structure into smaller directories, it improves the performance of looking up, creating, or deleting files, since these operations occur on smaller directories. The following sections summarize the design and implementation of Usenetfs. More detailed information is available in a separate report[23].

4.3.1 Design of Usenetfs

We had three design goals for Usenetfs. First, Usenetfs should not require changing existing news servers, operating systems, or file systems. Second, it should improve performance of these large directories enough to justify its overhead and complexity. Third, it should selectively manage large directories with little penalty to smaller ones.

The main idea for improving performance for large flat directories is to break them into smaller ones. Since article names are composed of sequential numbers, we take advantage of that. We create a hierarchy consisting of one thousand directories as depicted in Figure 4. We distribute articles across 1000 directories named 000 through 999. Since article numbers are sequential, we maximize the distribution by computing the final directory into which the article will go based on three of the four least significant digits. For example, the article named `control/cancel/123456` is placed into the directory `control/cancel/345/`. We picked the directory based on the second, third, and fourth digits of the article number to allow for some amount of clustering. By not using the least significant digit we cluster 10 consecutive arti-

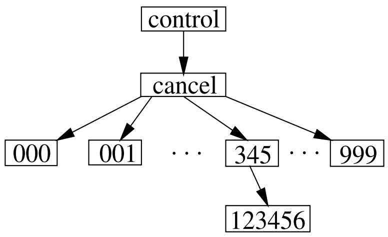

Figure 4: A Usenetfs Managed Newsgroup

cles together: the articles 123450–123459 are placed in the same directory. This increases the chance of kernel cache hits due to the likelihood of sequential access of these articles. In general, every article numbered X..XYYYZ is placed in a directory named YYY. For reading a whole directory (`readdir`), we iterate over the subdirectories from 000 to 999, and return the entries within.

Usenetfs needs to determine if a directory is managed or not. We co-opted a seldom used mode bit for directories, the setuid bit, to flag a directory as managed by Usenetfs. Using this bit lets news administrators control which directories are managed, using a simple `chmod` command.

The last issue was how to convert an unmanaged directory to be managed by Usenetfs: creating some of the 000–999 subdirectories and moving existing articles to their designated locations. Experimentally, we found that the number of truly large newsgroups is small, and that they rarely shrunk. Given that, and for simplicity, we made the process of turning directory management on/off an off-line process triggered by the news administrator with a provided script.

4.3.2 Implementation of Usenetfs

Usenetfs is the first non-trivial file system we designed and implemented using Wrapfs. By "non-trivial" we mean that it took us more than a few hours to achieve a working prototype from the Wrapfs template. It took us one day to write the first implementation, and several more days to test it and alternate restructuring algorithms (discussed elsewhere[23]).

We accomplished most of the work in the functions `encode_filename` and `decode_filename`. They check the setuid bit of the directory to see if it is managed by Usenetfs; if so, they convert the filename to its managed representation and back.

4.4 Cryptfs

Cryptfs is the most involved file system we designed and implemented based on Wrapfs. This section summarizes its design and implementation. More detailed information is available elsewhere[24].

We used the Blowfish[17] encryption algorithm—a 64 bit block cipher designed to be fast, compact, and simple. Blowfish is suitable in applications where the keys seldom change such as in automatic file decryptors. It can use variable length keys as long as 448 bits. We used 128 bit keys.

We picked the Cipher Block Chaining (CBC) encryption mode because it allows us to encrypt byte sequences of any length—suitable for encrypting file names. We decided to use CBC only within each encrypted block. This way ciphertext blocks (of 4–8KB) do not depend on previous ones, allowing us to decrypt each block independently. Moreover, since Wrapfs lets us manipulate file data in units of page size, encrypting them promised to be simple.

To provide stronger security, we encrypt file names as well. We do not encrypt "." and ".." to keep the lower level Unix file system intact. Furthermore, since encrypting file names may result in characters that are illegal in file names (nulls and "/"), we uuencode the resulting encrypted strings. This eliminates unwanted characters and guarantees that all file names consist of printable valid characters.

4.4.1 Key Management

Only the root user is allowed to mount an instance of Cryptfs, but can not automatically encrypt or decrypt files. To thwart an attacker who gains access to a user's account or to root privileges, Cryptfs maintains keys in an in-memory data structure that associates keys not with UIDs alone but with the combination of UID and session ID. To acquire or change a user's key, attackers would not only have to break into an account, but also arrange for their processes to have the same session ID as the process that originally received the user's passphrase. This is a more difficult attack, requiring session and terminal hijacking or kernel-memory manipulations.

Using session IDs to further restrict key access does not burden users during authentication. Login shells and daemons use `setsid(2)` to set their session ID and detach from the controlling terminal. Forked processes inherit the session ID from their parent. Users would normally have to authorize themselves only once in a shell. From this shell they could run most other programs that would work transparently and safely with the same encryption key.

We designed a user tool that prompts users for passphrases that are at least 16 characters long. The tool hashes the passphrases using MD5 and passes them to Cryptfs using a special `ioctl(2)`. The tool can also instruct Cryptfs to delete or reset keys.

Our design decouples key possession from file ownership. For example, a group of users who wish to edit a single file would normally do so by having the file group-owned by one Unix group and add each user to that group. Unix systems often limit the number of groups a user can be a member of to 8 or 16. Worse, there are often many subsets of users who are all members of one group and wish to share certain files, but are unable to guarantee the security of their shared files because there are other users who are members of the same group; e.g., many sites put all of their staff members in a group called "staff," students in the "student" group, guests in another, and so on. With our design, users can further restrict access to shared files only to those users who were given the decryption key.

One disadvantage of this design is reduced scalability with respect to the number of files being encrypted and shared. Users who have many files encrypted with different keys have to switch their effective key before attempting to access files that were encrypted with a different one. We do not perceive this to be a serious problem for two reasons. First, the amount of Unix file sharing of restricted files is limited. Most shared files are generally world-readable and thus do not require encryption. Second, with the proliferation of windowing systems, users can associate different keys with different windows.

Cryptfs uses one Initialization Vector (IV) per mount, used to jump-start a sequence of encryption. If not specified, a predefined IV is used. A superuser mounting Cryptfs can choose a different IV, but that will make all previously encrypted files undecipherable with the new IV. Files that use the same IV and key produce identical ciphertext blocks that are subject to analysis of identical blocks. CFS[2] is a user level NFS-based encryption file system. By default, CFS uses a fixed IV, and we also felt that using a fixed one produces sufficiently strong security.

One possible extension to Cryptfs might be to use different IVs for different files, based on the file's inode number and perhaps in combination with the page number. Other more obvious extensions to Cryptfs include the use of different encryption algorithms, perhaps different ones per user, directory, or file.

5 Performance

When evaluating the performance of the file systems we built, we concentrated on Wrapfs and the more complex file systems derived from Wrapfs: Cryptfs and Usenetfs. Since our file systems are based on several others, our measurements were aimed at identifying the overhead that each layer adds. The main goal was to prove that the overhead imposed by stacking is acceptably small and comparable to other stacking work[6, 18].

5.1 Wrapfs

We include comparisons to a native disk-based file system because disk hardware performance can be a significant factor. This number is the base to which other file systems compare to. We include figures for Wrapfs (our full-fledged stackable file system) and for lofs (the low-overhead simpler one), to be used as a base for evaluating the cost of stacking. When using lofs or Wrapfs, we mounted them over a local disk based file system.

To test Wrapfs, we used as our performance measure a full build of Am-utils[22], a new version of the Berkeley Amd automounter. The test auto-configures the package and then builds it. Only the sources for Am-utils and the binaries they create used the test file system; compiler tools were left outside. The configuration runs close to seven hundred small tests, many of which are small compilations and executions. The build phase compiles about 50,000 lines of C code in several dozen files and links eight binaries. The procedure contains both CPU and I/O bound operations as well as a variety of file system operations.

For each file system measured, we ran 12 successive builds on a quiet system, measured the elapsed times of each run, removed the first measure (cold cache) and averaged the remaining 11 measures. The results are summarized in Table 3.[5] The standard deviation for the results reported in this section did not exceed 0.8% of the mean. Finally, there is no native lofs for FreeBSD, and the nullfs available is not fully functional (see Section 3.6).

File System	**SPARC 5**		**Intel P5/90**		
	Solaris 2.5.1	**Linux 2.0.34**	**Solaris 2.5.1**	**Linux 2.0.34**	**FreeBSD 3.0**
ext2/ufs/ffs	1242.3	1097.0	1070.3	524.2	551.2
lofs	1251.2	1110.1	1081.8	530.6	n/a
wrapfs	1310.6	1148.4	1138.8	559.8	667.6
cryptfs	1608.0	1258.0	1362.2	628.1	729.2
crypt-wrap	22.7%	9.5%	19.6%	12.2%	9.2%
nfs	1490.8	1440.1	1374.4	772.3	689.0
cfs	2168.6	1486.1	1946.8	839.8	827.3
cfs-nfs	45.5%	3.2%	41.6%	8.7%	20.1%
crypt-cfs	34.9%	18.1%	42.9%	33.7%	13.5%

Table 3: Time (in seconds) to build a large package on various file systems and platforms. The percentage lines show the overhead difference between some file systems

First we evaluate the performance impact of stacking a file system. Lofs is 0.7–1.2% slower than the native disk based file system. Wrapfs adds an overhead of 4.7–6.8% for Solaris and Linux systems, but that is comparable to the 3–10% degradation previously reported for null-layer stackable file systems[6, 18]. On FreeBSD, however, Wrapfs adds an overhead of 21.1% compared to FFS: to overcome limitations in nullfs, we used synchronous writes. Wrapfs is more costly than lofs because it stacks over every vnode and keeps its own copies of data, while lofs stacks only on directory vnodes, and passes all other vnode operations to the lower level verbatim.

5.2 Cryptfs

Using the same tests we did for Wrapfs, we measured the performance of Cryptfs and CFS[2]. CFS is a user level NFS-based encryption file system. The results are also summarized in Table 3, for which the standard deviation did not exceed 0.8% of the mean.

Wrapfs is the baseline for evaluating the performance impact of the encryption algorithm. The only difference between Wrapfs and Cryptfs is that the latter encrypts and decrypts data and file names. The line marked as "*crypt-wrap*" in Table 3 shows that percentage difference between Cryptfs and Wrapfs for each platform. Cryptfs adds an overhead of 9.2–22.7% over Wrapfs. That significant overhead is unavoidable. It is the cost of the Blowfish cipher, which, while designed to be fast, is still CPU intensive.

[5] All machines used in these tests had 32MB RAM.

Measuring the encryption overhead of CFS was more difficult. CFS is implemented as a user-level NFS file server, and we also ran it using Blowfish. We expected CFS to run slower due to the number of additional context switches that it incurs and due to NFS v.2 protocol overheads such as synchronous writes. CFS does *not* use the NFS server code of the given operating system; it serves user requests directly to the kernel. Since NFS server code is implemented in general inside the kernel, it means that the difference between CFS and NFS is not just due to encryption, but also due to context switches. The NFS server in Linux 2.0 is implemented at user-level, and is thus also affected by context switching overheads. If we ignore the implementation differences between CFS and Linux's NFS, and just compare their performance, we see that CFS is 3.2–8.7% slower than NFS on Linux. This is likely to be the overhead of the encryption in CFS. That overhead is somewhat smaller than the encryption overhead of Cryptfs because CFS is more optimized than our Cryptfs prototype: CFS precomputes large stream ciphers for its encrypted directories.

We performed microbenchmarks on the file systems listed in Table 3 (reading and writing small and large files). These tests isolate the performance differences for specific file system operations. They show that Cryptfs is anywhere from 43% to an order of magnitude faster than CFS. Since the encryption overhead is roughly 3.2–22.7%, we can assume that rest of the difference comes from the reduction in number of context switches. Details of these additional measurements are available elsewhere[24].

5.3 Usenetfs

We configured a News server consisting of a Pentium-II 333Mhz, with 64MB of RAM, and a 4GB fast SCSI disk for the news spool. The machine ran Linux 2.0.34 with our Usenetfs. We created directories with exponentially increasing numbers of files in each: 1, 2, 4, etc. The largest directory had 524288 (2^{19}) files numbered starting with 1. Each file was 2048 bytes long. This size is the most common article size on our production news server. We created two hierarchies with increasing numbers of articles in different directories: one flat and one managed by Usenetfs.

We designed our next tests to match the two actions most commonly undertaken by a news server (see Table 2). First, a news server looks up and reads articles, mostly in response to users reading news and when processing outgoing feeds. The more users there are, the more random the article numbers read tend to be. While users read articles in a mostly sequential order, the use of threaded newsreaders results in more random reading. The (log-log) plot of Figure 5 shows the performance of 1000 random lookups in both flat and Usenetfs-managed directories. The times reported are in milliseconds spent by the process and the operating system on its behalf. For random lookups on directories with fewer than 1000–2000 articles, Usenetfs

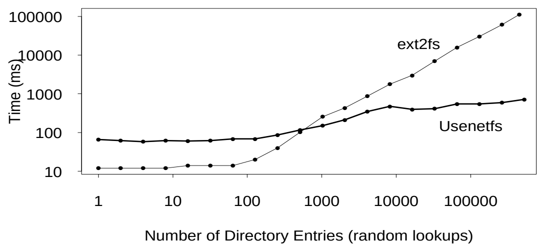

Figure 5: Cost for 1000 Random Article Lookups

adds overhead and slows performance. We expected this because the bushier directory structure Usenetfs maintains has over 1000 subdirectories. As directory sizes increase, lookups on flat directories become linearly more expensive while taking an almost constant time on Usenetfs-managed directories. The difference exceeds an order of magnitude for directories with over 10,000 articles.

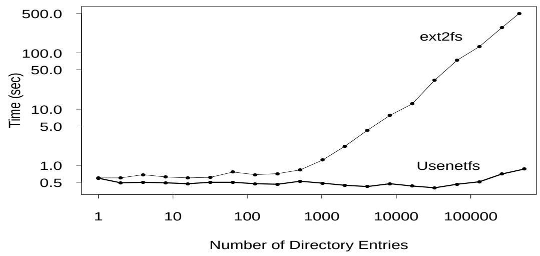

Figure 6: Cost for 1000 Article Additions and Deletions

The second common action a news server performs is creating new articles and deleting expired ones. New articles are created with monotonically increasing numbers. Expired articles are likely to have the smallest numbers so we made that assumption for the purpose of testing. Figure 6 (also log-log) shows the time it took to add 1000 new articles and then remove the 1000 oldest articles for successively increasing directory sizes. The results are more striking here: Usenetfs times are almost constant throughout, while adding and deleting files in flat directories took linearly increasing times.

Creating over 1000 additional directories adds overhead to file system operations that need to read whole directories, especially the readdir call. The last Usenetfs test takes into account all of the above factors, and was performed on our departmental production news server. A simple yet realistic measure of the overall performance of the system is to test how much reserve capacity was left in the server. We tested that by running a repeated set of compilations of a large package (Am-utils), timing how long it took to complete each build. We measured the compile times of Am-utils, once when the news server was running without Usenetfs management, and then when Usenetfs managed the top 6 newsgroups. The results are depicted in Figure 7. The average compile time was reduced by 22% from 243 seconds to 200 seconds. The largest savings appeared during busy times when our server transferred outgoing articles to our upstream feeds, and especially during the four daily expiration periods. During these expiration peaks, performance improved by a factor of 2–3. The overall effect of Usenetfs had been to keep the performance of the news server more flat, removing those load surges. The standard deviation for the compiles was reduced from 82 seconds (34% of the mean) to 61 seconds (29% of the mean). Additional performance analysis is provided elsewhere[23].

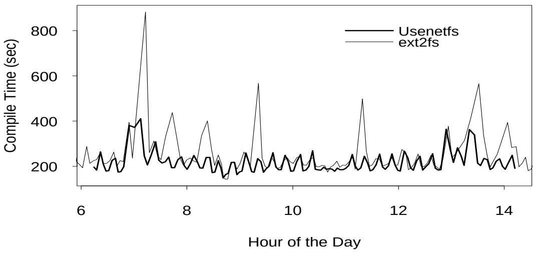

Figure 7: Compile Times on a Production News Server

5.4 Lb2fs

Lb2fs's performance is less than 5% slower than Wrapfs. The two main differences between Wrapfs and Lb2fs are the random selection algorithm and looking up directory vnodes on both replicas. The impact of the random selection algorithm is negligible, as it picks the least-significant bit of an internal system clock. The impact of looking up directory vnodes twice is bound by the ratio of directories to non-directories in common shared file systems. We performed tests at our department and found that the number of directories in such file systems to be 2–5% of the overall number of files. That explains the small degradation in performance of Lb2fs compared to Wrapfs.

5.5 Portability

We first developed Wrapfs and Cryptfs on Solaris 2.5.1. As seen in Table 4, it took us almost a year to fully develop Wrapfs and Cryptfs together for Solaris, during which time we had to overcome our lack of experience with Solaris kernel internals and the principles of stackable file systems. As we gained experience, the time to port the same file system to a new operating system grew significantly shorter. Developing these file systems for Linux 2.0 was a matter of days to a couple of weeks. This port would have been faster had it not been for Linux's different vnode interface.

File System	Solaris 2.x	Linux 2.0	FreeBSD 3.0	Linux 2.1
wrapfs	9 months	2 weeks	5 days	1 week
snoopfs	1 hour	1 hour	1 hour	1 hour
lb2fs	2 hours	2 hours	2 hours	2 hours
usenetfs		4 days		1 day
cryptfs	3 months	1 week	2 days	1 day

Table 4: Time to Develop and Port File Systems

The FreeBSD 3.0 port was even faster. This was due to many similarities between the vnode interfaces of Solaris and FreeBSD. We recently also completed these ports to the Linux 2.1 kernel. The Linux 2.1 vnode interface made significant changes to the 2.0 kernel, which is why we list it as another porting effort. We held off on this port until the kernel became more stable (only recently).

Another metric of the effort involved in porting Wrapfs is the size of the code. Table 5 shows the total number of source lines for Wrapfs, and breaks it down to three categories: common code that needs no porting, code that is easy to port by simple inspection of system headers, and code that is difficult to port. The hard-to-port code accounts for more than two-thirds of the total and is the one involving the implementation of each Vnode/VFS operation (operating system specific).

Porting Difficulty	Solaris 2.x	Linux 2.0	FreeBSD 3.0	Linux 2.1
Hard	80%	88%	69%	79%
Easy	15%	7%	26%	10%
None	5%	3%	5%	11%
Total Lines	3431	2157	2882	3279

Table 5: Wrapfs Code Size and Porting Difficulty

The difficulty of porting file systems written using Wrapfs depends on several factors. If plain C code is used in the Wrapfs API routines, the porting effort is minimal or none. Wrapfs, however, does not restrict the user from calling any in-kernel operating system specific function. Calling such functions complicates portability.

6 Related Work

Vnode stacking was first implemented by Rosenthal (in SunOS 4.1) around 1990[15]. A few other works followed Rosenthal, such as further prototypes for extensible file systems in SunOS[18], and the Ficus layered file system[4, 7] at UCLA. Webber implemented file system interface extensions that allow user level file servers[20]. Unfortunately this work required modifications to existing file systems and could not perform as well as in-kernel file systems.

Several newer operating systems offer a stackable file system interface. They have the potential of easy development of file systems offering a wide range of services.

Their main disadvantages are that they are not portable enough, not sufficiently developed or stable, or they are not available for common use. Also, new operating systems with new file system interfaces are not likely to perform as well as ones that are several years older.

The *Herd of Unix-Replacing Daemons* (HURD) from the Free Software Foundation (FSF) is a set of servers running on the Mach 3.0 microkernel[1] that collectively provide a Unix-like environment. HURD file systems run at user level. HURD introduced the concept of a translator, a program that can be attached to a pathname and perform specialized services when that pathname is accessed. Writing a new translator is a matter of implementing a well defined file access interface and filling in such operations as opening files, looking up file names, creating directories, etc.

Spring is an object-oriented research operating system built by Sun Microsystems Laboratories[10]. It was designed as a set of cooperating servers on top of a microkernel. Spring provides several generic modules which offer services useful for a file system: caching, coherency, I/O, memory mapping, object naming, and security. Writing a file system for Spring entails 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[8]. 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 the benefits that come with Spring, while requiring little or no change to existing file systems; those can be gradually ported 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.

7 Conclusions

Wrapfs and the examples here prove that useful, non-trivial vnode stackable file systems can be implemented on modern operating systems without having to change the rest of the system. We achieve better performance by running the file systems in the kernel instead of at user-level. File systems built from Wrapfs are more portable than other kernel-based file systems because they interact directly with a (mostly) standard vnode interface.

Most complications discovered while developing Wrapfs stemmed from two problems. First, the vnode interface is not self-contained; the VM system, for example, offers memory mapped files, but to properly handle them we had to manipulate lower level file systems and MMU/TLB hardware. Second, several vnode calls (such as `readdir`) are poorly designed.

Estimating the complexity of software is a difficult task. Kernel development in particular is slow and costly because of the hostile development environment. Furthermore, personal experience of the developers figure heavily in the cost of development and testing of file systems. Nevertheless, it is our assertion that once Wrapfs is ported to a new operating system, other non-trivial file systems built from it can be prototyped in a matter of hours or days. We estimate that Wrapfs can be ported to any operating system in less than one month, as long as it has a vnode interface that provides a private opaque field for each of the major data structures of the file system. In comparison, traditional file system development often takes a few months to several years.

Wrapfs saves developers from dealing with kernel internals, and allows them to concentrate on the specifics of the file system they are developing. We hope that with Wrapfs, other developers could prototype new file systems to try new ideas, develop fully working ones, and port them to various operating systems—bringing the complexity of file system development down to the level of common user-level software.

We believe that a truly stackable file system interface could significantly improve portability, especially if adopted by the main Unix vendors. We think that Spring[10] has a very suitable interface. If that interface becomes popular, it might result in the development of many practical file systems.

7.1 Future

We would like to add to Wrapfs an API for manipulating file attributes. We did not deem it important for the initial implementation because we were able to manipulate the attributes needed in one place anyway.

Wrapfs cannot properly handle file systems that change the size of the file data, such as with compression, because these change file offsets. Such a file system may have to arbitrarily shift data bytes making it difficult to manipulate the file in fixed data chunks. We considered several designs, but did not implement any, because they would have complicated Wrapfs's code too much, and would mostly benefit compression.

8 Acknowledgments

The authors thank the anonymous reviewers and especially Keith Smith, whose comments improved this paper significantly. We would also like to thank Fred Korz, Seth Robertson, Jerry Altzman, and especially Dan Duchamp for their help in reviewing this paper and offering concrete suggestions. 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] M. Blaze. A Cryptographic File System for Unix. *Proc. of the first ACM Conf. on Computer and Communications Security*, November 1993.

[3] B. Callaghan and S. Singh. The Autofs Automounter. *USENIX Conf. Proc.*, pages 59–68, Summer 1993.

[4] 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, June 1990.

[5] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

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

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

[8] V. Matena, Y. A. Khalidi, and K. Shirriff. Solaris MC File System Framework. Technical Report TR-96-57. Sun Labs, October 1996.

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

[10] J. G. Mitchel, J. J. Giobbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. *CompCon Conf. Proc.*, 1994.

[11] 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, June 1994.

[12] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. *USENIX Conf. Proc.*, pages 25–33, January 1995.

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

[14] D. S. H. Rosenthal. Requirements for a “Stacking” Vnode/VFS Interface. Unix International document SD-01-02-N014. 1992.

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

[16] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Conf. Proc.*, pages 119–30, June 1985.

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

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

[19] SMCC. lofs – loopback virtual file system. SunOS 5.5.1 Reference Manual, Section 7. March 1992.

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

[21] E. Zadok. *FiST: A File System Component Compiler*. PhD thesis, published as Technical Report CUCS-033-97. Computer Science Department, Columbia University, April 1997.

[22] E. Zadok. Am-utils (4.4BSD Automounter Utilities). Am-utils version 6.0a16 User Manual. April 1998. Available http://www.cs.columbia.edu/~ezk/am-utils/.

[23] E. Zadok and I. Badulescu. Usenetfs: A Stackable File System for Large Article Directories. Technical Report CUCS-022-98. Computer Science Department, Columbia University, June 1998.

[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, July 1998.

9 Author Information

Erez Zadok is an Ph.D. candidate in the Computer Science Department at Columbia University. He received his B.S. in Comp. Sci. in 1991, and his M.S. degree in 1994, both from Columbia University. His primary interests include file systems, operating systems, networking, and security. The work described in this paper was first mentioned in his Ph.D. thesis proposal[21].

Ion Badulescu holds a B.A. from Columbia University. His primary interests include operating systems, networking, compilers, and languages.

Alex Shender is the manager of the computer facilities at Columbia University's Computer Science Department. His primary interests include operating systems, networks, and system administration. In May 1998 he received his B.S. in Comp. Sci. from Columbia's School of Engineering and Applied Science.

For access to sources for the file systems described in this paper see *http://www.cs.columbia.edu/~ezk/research/software/*.