On Incremental File System Development

EREZ ZADOK, RAKESH IYER, NIKOLAI JOUKOV, GOPALAN SIVATHANU, AND CHARLES P. WRIGHT

Developing file systems from scratch is difficult and error prone. Layered, or stackable, file systems are a powerful technique to incrementally extend the functionality of existing file systems on commodity OSes at runtime. In this paper, we analyze the evolution of layering from historical models to what is found in four different present day commodity OSes: Solaris, FreeBSD, Linux, and Microsoft Windows. We classify layered file systems into five types based on their functionality and identify the requirements that each class imposes on the OS. We then present five major design issues that we encountered during our experience of developing over twenty layered file systems on four OSes. We discuss how we have addressed each of these issues on current OSes, and present insights into useful OS and VFS features that would provide future developers more versatile solutions for incremental file system development.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Portability*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; D.4.3 [**Operating Systems**]: File Systems Management—*File organization*; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design

Additional Key Words and Phrases: Layered File Systems, Stackable File Systems, VFS, Vnode, I/O Manager, IRP, Extensibility

1. INTRODUCTION

Data management is a fundamental facility provided by the operating system (OS). File systems are tasked with the bulk of data management, including storing data on disk (or over the network) and naming (i.e., translating a user-visible name such as `/usr/src` into an on-disk object). File systems are complex, and it is difficult to enhance them. Furthermore, OS vendors are reluctant to make major changes to a file system, because file system bugs have the potential to corrupt all data on a machine. Because file system development is so difficult, extending file system functionality in an incremental manner is valuable. Incremental development also makes it possible for a third-party software developer to release file system improvements, without developing a whole file system from scratch.

Originally, file systems were thoroughly integrated into the OS, and system calls directly invoked file system methods. This architecture made it difficult to add multiple file systems. The introduction of a *virtual node* or *vnode* provided a layer of abstraction that separates the core of the OS from file systems [Kleiman 1986]. Each file is represented in memory by a vnode. A vnode has an operations vector that defines several operations that the OS can call, thereby allowing the OS to add and remove types of file systems at runtime. Most current OSes use something similar to the vnode interface, and the number of file systems supported by the OS has grown accordingly. For example, Linux 2.6 supports over 30 file systems and many more are maintained outside of the official kernel tree.

Clearly defining the interface between the OS and file systems makes interposition possible. A *layered*, or *stackable*, file system creates a vnode with its own operations vector to be interposed on another vnode. Each time one of the layered file system's operations is invoked, the layered file system maps its own vnode to a lower-level vnode, and then calls the lower-level vnode's operation. To add functionality, the layered file system can perform additional operations before or after the lower-level operation (e.g., encrypting data before a `write` or decrypting data after a `read`). The key advantage of layered file systems is that they can change the functionality of a commodity OS at runtime so hard-to-develop lower-level file systems do not need to be changed. This is important, because OS developers often resist change, especially to file systems where bugs can cause data loss.

Rosenthal was among the first to propose layering as a method of extending file systems [Rosenthal 1990; 1992]. To enable layering, Rosenthal radically changed the VFS internals of SunOS. Each public vnode field was converted into a method; and all knowledge of vnode types (e.g., directory vs. regular file) was removed from the core OS. Researchers at UCLA independently developed another layering infrastructure [Heidemann and Popek 1991; 1994] that placed an emphasis on light-weight layers and extensibility. The original pioneers of layering envisioned creating building blocks that could be composed together to create more sophisticated and rich file systems. For example, the directory-name lookup cache (DNLC) could simply be implemented as a file system layer, which returns results on a cache hit, but passes operations down on a miss [Skinner and Wong 1993].

Layering has not commonly been used to create and compose building-block file systems, but instead has been widely used to add functionality rapidly and portably to existing file systems. Many applications of layered file system are features that could be implemented as part of the VFS (e.g., unification), but for practical reasons it is easier to develop them as layered file systems. Several OSes have been designed to support layered file systems, including Solaris, FreeBSD, and Windows. Several layered file systems are available for Linux, even though it was not originally designed to support them. Many users use layered file systems unknowingly as part of Antivirus solutions [Symantec 2004; Miretskiy et al. 2004], and Windows XP's system restore feature [Harder 2001]. On Unix, a null-layer file system is used to provide support for accessing one directory through multiple paths. When the layer additionally modifies the data, useful new functionality like encryption [Corner and Noble 2002; Halcrow 2004] or compression [Zadok et al. 2001] can be added. Another class of layered file systems, called *fan out*, operates directly on top of several lower-level file systems. For example, unification file systems merge the contents of several directories [Pendry and McKusick 1995; Wright et al. 2006]. Fanout file systems can also be used for replication, load-balancing, failover, snapshotting, and caching.

The authors of this paper have over fifteen years of combined experience developing layered file systems on four OSes: Solaris, FreeBSD, Linux, and Windows. We have developed more than twenty layered file systems that provide encryption, compression, versioning, tracing, antivirus, unification, snapshotting, replication, checksumming, and more.

The rest of this paper is organized as follows. In Section 2 we survey alternative techniques to enhance file system functionality. In Section 3 we describe four models of layered file system development. We then proceed to describe five broad classes of layered file systems in Section 4. In Section 5 we describe five general problems and their solutions that are useful for all types of layered file systems. We conclude in Section 6 with guiding principles for future OS and layered file system developers.

2. RELATED WORK

In this section we describe alternatives to achieve the extensibility offered by layered file systems. We discuss four classes of related works based on the level at which extensibility is achieved: in hardware, in the device driver, at the system-call level, or in user-level programs. We have a detailed discussion of layered file system infrastructures in Section 3.

Hardware level. Slice [Anderson et al. 2000] is a storage system architecture for high speed networks with network-attached block storage. Slice interposes a piece of code called a *switching filter* in the network hardware to route packets among a group of servers. Slice appears to the upper level as a single block-oriented storage device. High-level control information (e.g., files) is unavailable to interposition code at the hardware level, and therefore cannot perform optimizations for specific devices.

Semantically-Smart Disk Systems (SDSs) [Sivathanu et al. 2003] attempt to provide file-system–like functionality without modifying the file system. Knowledge of a specific file system is embedded into the storage device, and the device provides additional functionality that would traditionally be implemented in the file system. Such systems are relatively easy to deploy, because they do not require modifications to existing file system code. Layered file systems share a similar goal in terms of reusing and leveraging existing infrastructures. Unlike a layered file system, an SDS is closely tied to the format of the file system running on top of it, so porting SDSs to new file systems is difficult.

Device-driver level. Software RAID and Logical Volume Managers (LVMs) introduce another layer of abstraction between the storage hardware and the file system. They provide additional features such as increased reliability and performance, while appearing to the file system as a simple disk, which makes them easy to deploy in existing infrastructure. For example, on Linux a Cryptoloop devices uses a loopback block driver to encrypt data stored on a disk or in a file. A new file system is then created within the Cryptoloop device. Any file system can run on top of a block device-driver extension. However, block device extensions cannot exploit the control information (e.g., names) that is available at the file system level.

System-call level. SLIC [Ghormley et al. 1998] is a protected extensibility system for OSes that uses interposition techniques to enable the addition of a large class of untrusted extensions to existing code. Several OS extensions can be implemented using SLIC such as encryption file systems and a protected environment for binary execution. The Interposition Agents toolkit [Jones 1993], developed by Microsoft Research, allows a user's code to be written in terms of high-level objects provided by this interface. The toolkit was designed to ease interposing code between the clients and the instances of the system interface to facilitate adding new functionality like file reference tracing, customizable file system views, etc. to existing systems. Similarly, Mediating Connectors [Balzer and Goldman 1999] is a system call (and library call) wrapper mechanism for Windows NT that allows users to trap API calls.

System call interposition techniques rely on the communication channel between user-space and the kernel, and hence cannot handle operations that bypass that channel (e.g., `mmap` operations and their associated page faults). Also, interposing at the system call level results in overhead for all system calls even if only a subset of kernel components (e.g., the file system) need to be interposed.

User level. Gray-box Information and Control Layers (ICL) [Arpaci-Dusseau and Arpaci-Dusseau 2001] extend the functionality of OSes by acquiring information about their internal state. ICLs provide OS-like functionality without modifying existing OS code. Dust [Burnett et al. 2002] is a direct application of ICLs that uses gray-box knowledge of the OS's buffer cache management policy to optimize application performance. For example, if a Web server first services Web pages that are believed to be in the OS buffer cache, then both average response time and throughput can be improved.

Blaze's CFS is a cryptographic file system that is implemented as a user-level NFS server [Blaze 1993]. The OS's unmodified NFS client mounts the NFS server over the loopback network interface. The SFS toolkit [Maziéres 2001] aims to simplify Unix file system extensibility by allowing development of file systems at the user level. Using the toolkit, one can implement a simple user-level NFS server and redirect local file system operations into the user level implementation. The popularity of the SFS toolkit demonstrates that developers have observed the complexity of modifying existing time-tested file systems. SiRiUS [Goh et al. 2003], a file system for securing remote untrusted storage, and Dabek's CFS [Dabek et al. 2001], a wide area cooperative file system, were built using the SFS toolkit.

Filesystem in Userspace [Szeredi 2005], or FUSE, is a hybrid approach that consists of two parts: (1) a standard kernel-level file system which passes calls to a user-level demon, and (2) a library to easily develop file-system–specific FUSE demons. Developing new file systems with FUSE is relatively simple because the user-level demon can issue normal system calls (e.g., `read`) to service a VFS call (e.g., `vfs_read`). The main two disadvantages of a FUSE file system are that (1) performance is limited by crossing the user-kernel boundary, and (2) the file system can only use FUSE's API, which closely matches the VFS API, whereas kernel file systems may access a richer kernel API (e.g., for process and memory management).

Sun designed and developed Spring as an object-oriented microkernel OS. Spring's architecture allowed various components, including file systems, to be transparently extended with user libraries [Khalidi and Nelson 1993]. Spring's design was radically different from current commodity OSes. As it was research prototype, it was not deployed in real systems. K42 [Appavoo et al. 2002] is a new OS under development at IBM which incorporates innovative mechanisms and modern programming technologies. Various system functionalities can be extended at the user level through libraries by providing a microkernel-like interface. The Exokernel architecture [Engler et al. 1995] implements minimal components in the kernel and allows user-level applications to customize OS functionality using library OSes.

In general, user-level extensions are easy to implement, but their performance is not as good as kernel extensions because the former involve data copies between the user level and the kernel level, as well as additional context switches.

3. LAYERING MODELS

On Unix, the idea of layering evolved from the vnode interface [Kleiman 1986], initially implemented in SunOS in 1984. Most of today's layered file system models use a vnode interface. Microsoft Windows, on the other hand, uses a message-passing layering model. We describe three Unix models and the Windows model here; these cover the full range of popular or commodity models available. In Section 3.1 we describe Rosenthal's object-oriented model. In Section 3.2 we describe the model contemporaneously developed at UCLA. In Section 3.3 we describe the layering model found in three modern Unix systems (Linux, FreeBSD, and Solaris). In Section 3.4 we describe the Windows 2000 and Windows XP message-passing layering model. We present a summary table of the features offered by each layering model in Section 3.5.

3.1 Rosenthal's Layering Model

In 1990, Rosenthal developed an experimental prototype of a new VFS for SunOS [Rosenthal 1990] with the goal of layering vnodes, so that file systems can be composed of building blocks. Rosenthal identified two distinct types of layering. The first, shown in Figure 1(a), is *interposition* in which a higher-level vnode is called before the lower-level vnode, and can modify the lower-level vnode's arguments, operation, and results. Today, this is commonly called *linear layering* or *linear stacking*. The second, shown in Figure 1(b) is *composition* in which a higher-level vnode performs operations on several lower-level vnodes. Today, this is commonly called *fan out*.

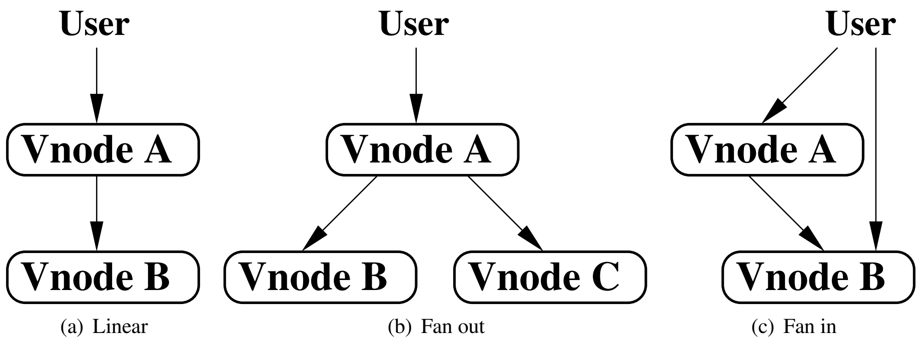

Fig. 1. Types of layering. In a linear layer all operations are intercepted by the top-level vnode, A, and A passes it to a single vnode below. In fan-out, all operations go through the top-level vnode, A, but there are two or more lower-level vnodes. In fan-in, some operations go through the top-level vnode, A, before going to the lower-level vnode, B, but some operations can bypass A and directly access B.

Rosenthal introduced two key abstractions to support vnode layering: *push* and *pop* for inserting and removing vnodes from a stack. All vnode operations go through the highest-level vnode. Rosenthal replaced all visible attributes in the vnode structure with methods so that layered file systems could intercept them. Push inserts a vnode at the top of a stack of vnodes, so that all operations destined for the stack of vnodes go to the new top vnode. The top vnode may in turn call the vnodes that are below it. Pop performs the opposite operation: the top vnode from a stack is removed and operations go directly to the lower-level vnode. Two new fields were added to the vnode structure: `v_top` and `v_above`.

The `v_above` pointer points to the vnode that is directly above this one in the stack. The `v_top` pointer points to the highest level vnode in the stack. All vnode operations go through the highest-level vnode, hence every vnode in the stack essentially becomes an alias to the highest-level vnode. Unfortunately, this prevents fan-in access (shown in Figure 1(c)), in which a user process accesses the lower-level file system directly. Fan-in access is necessary when applications need to access unmodified data; for example, a backup program should write encrypted data (stored on the lower-level file system) to tape, not the unencrypted data (accessible through the top layer). Rosenthal also suggested several layered file system building blocks, but did not implement any of them for his prototype.

Rosenthal proposed a set of requirements for a layered vnode interface [Rosenthal 1992]. To support interposition, he concluded that the OS must support replacing a vnode's operations vector and private data for each interposer. Moreover, all vnode operations must be mitigated through the operations vector. To support composition, Rosenthal concluded that vnodes must support transactions. The vnode interfaces assumes that each operation is atomic [Kleiman 1986]. However, to compose two file systems together, two distinct lower-level vnode operations must be called; thus the overall operation is not atomic.

Later, a vnode interface for SunOS based on Rosenthal's interposition and several example file systems were developed [Skinner and Wong 1993]. Of particular note is that some existing VFS operations took multiple vnode arguments, making it impossible to determine which vnode's operations vector should handle the operation. For example, `link` would take two vnode arguments, the parent directory and the vnode to insert into that directory. To make each vnode control all of its operations, Skinner and Wong decomposed these larger vnode operations into smaller operations to manage vnode link counts, to update directory entries, and to create objects. Skinner and Wong's prototype dealt with issues that Rosenthal did not, including concurrency, locking, and per-mount operations. They also developed file system components in a layered manner. Skinner and Wong's implementation of `mount` and `unmount` is a particularly elegant example of push and pop. The OS's name-lookup code no longer needs special code for mount points. The root vnode of the file system to be mounted is simply pushed on top of the covered vnode. To unmount a file system, its root vnode is simply popped from the stack. The directory-name–lookup cache was also implemented as an interposition layer.

3.2 UCLA's Layering Model

In the early 1990s, Heidemann and Popek also developed an infrastructure for layered file system development at UCLA [Heidemann and Popek 1991; 1994]. The UCLA model emphasized general VFS extensibility and the ability to compose many small and efficient building block layers into more sophisticated file system services. The UCLA model suggests that each separate service should be its own layer. For example, UFS should be divided into at least three different layers: (1) managing disk partitions, (2) a file storage layer providing arbitrary-length files referenced by a unique identifier (i.e., inode number), and (3) a hierarchical directory component. Breaking up a file system like UFS would allow interposition at several points. For example, an intrusion-detection layer could be inserted above the directory component so it has access to names, but a compression layer could be inserted between the naming layer and the file storage layer to avoid the implementation details of hard links.

To provide extensibility, the UCLA model does not have a fixed set of operations. Instead, each file system provides its own set of operations, and the total set of operations is the union of all file systems' operations. If a file system does not support a given operation, then a generic routine for that file system is called. This routine could, for example, simply return an "Operation not supported" error. However, for a layered file system, a generic bypass routine can be used. To provide operations with arbitrary arguments, the UCLA interface transforms all of an operation's arguments into a single structure of arguments. Along with the structure, meta-data is passed to the operation that identifies each argument's offset and type. Using this meta-data, the bypass routine can generically map its own vnodes to lower-level vnodes and invoke the lower-level operation.

The UCLA model uses mount and unmount to instantiate a layered file system. The existing mount operation fits well for two reasons: (1) it creates a new subtree of objects within the file system name space, and (2) creation of subtrees usually requires an existing file system object (e.g., to mount a UFS file system you need a device node). A file-system layer often accesses an existing directory in much the same way that UFS uses a device. For example, to mount an encryption file system, the path to the encrypted data is used.

Caching is essential to provide good performance, but can pose problems if two or more layers cache an object. For example, if two layers concurrently cache the same page or vnode, and one modifies it, then the other one would read stale data. Similarly, if both of them modify the page or vnode, then one of them would lose its update. Heidemann developed a prototype general-purpose cache manager that provided coherent access to all file system layers [Heidemann and Popek 1995]. The UCLA cache manager requires layers to request either shared or exclusive *ownership* of an object before caching it. When required, the cache manager revokes the existing layers' ownership using a callback. In most cases, the cache manager automatically determines when two objects represent the same data. However, if layers may change the semantics of the data, then the cache manager cannot correlate the upper-level and lower-level file's position. In these cases, the cache manager treats the upper-level and lower-level objects as two distinct objects, and all ownership requests for either object are sent to the layer that changes the semantics. The semantic-changing layer can then invoke the correct callbacks based on its knowledge of the relationship between the original and transformed data.

The Ficus replicated file system and several course projects (e.g., encryption and compression layers) were developed using the UCLA model [Guy et al. 1990].

3.3 Layering in Modern Unix Systems

In this section we describe layering in three modern Unix-based OSes: Solaris, FreeBSD, and Linux. We discuss their pros and cons in terms of ease of development and performance. All three systems use the vnode interface to enable layering functionality, but they have key implementation differences that influence development ease and performance.

Solaris. The architecture of the Solaris VFS is nearly identical to the classic vnode architecture. Each vnode has a fixed operations vector. Each operation must be implemented by every file system (undefined operations are not permitted). Generic operations for returning "operation not supported" or in some cases success are available. Mutable attributes such as size, access time, etc. are not part of vnode fields; instead, the vnode operations include functions for managing such attributes.

The Solaris loopback file system, Lofs [SMCC 1992], is a null pass-through layer that layers only on directory vnodes. Thus, for regular files on Lofs, vnode objects are not duplicated. Because Lofs does not layer on file vnodes, the data pages of files are not duplicated, resulting in more efficient memory utilization. However, this makes it harder to extend Lofs to provide functionality like encryption, where two different page contents need to be maintained at the lower level and the layered level.

CacheFS [SunSoft 1994] is a layered fan-out file system in Solaris which can mount on top of two lower level directories, the source and the cache directory. When files in the source directory are accessed through CacheFS, they are copied into the root of the cache directory and indexed using a numeric identifier.

FreeBSD. The FreeBSD vnode interface has extensibility based on the UCLA model. FreeBSD allows dynamic addition of vnode operations. While activating a file system, the kernel registers the set of vnode operations that the file system supports, and builds an operation vector for each file system that contains the union of all operations supported by any file system. File systems provide default routines for operations that they do not support. FreeBSD uses packed arguments so layers can generically bypass operations as in the UCLA model.

FreeBSD's version of the loopback file system is called *nullfs* [Pendry and McKusick 1995]. It is a simple file system layer and makes no transformations on its arguments, just passing the requests it receives to the lower file system. FreeBSD's union mounts [Pendry and McKusick 1995] logically merge the directories that they are mounted on.

Compared to Solaris (and Linux), FreeBSD has the most versatile support for layering at the file system level. Its architecture allows new vnode operations to be created dynamically. Packing function arguments in an argument structure allows layers to bypass operations that they do not need to intercept easily.

Linux. Linux does not have native support for layered file systems. The Linux VFS was designed to make adding file systems relatively simple by separating generic file system code from the individual file system implementations. A file system may choose not to implement a method, and the VFS itself provides a sensible default. For example, a file system can use generic routines for most data-oriented operations (i.e., reading from and writing to files, including via `mmap`), and only needs to provide two primitive operations to read and write a block from the file. This trend of extracting more and more abstractions is on the rise in the new versions of the Linux kernel. Layered file systems are usually not able to use generic methods, because they need to pass the operation to the lower-level file system, but generic operations do not make it more difficult to develop layered file systems. On the other hand, providing functionality directly within the VFS makes it more difficult to implement layered file systems. As shown in Figure 2, a layered file system must appear just like the VFS to the lower-level file system [Zadok and Bădulescu 1999], so it must replicate this VFS functionality. Worse, as the VFS changes, the layered file system must also change, or subtle bugs could be introduced.

On Linux, the vnode object is divided into two components: a *dentry* which represents the name of the file and an *inode* which represents the actual on-disk file. This separation simplifies support for hard links. On Linux, VFS operation vectors are fixed. Operations cannot be added, and the prototype of existing operations cannot be changed. The Linux VFS stores mutable values inside the inode object. This makes it more difficult to maintain cache coherency between the inodes of the layered file system and the lower level inodes. It also causes performance degradation in layered file systems as the modified values in the

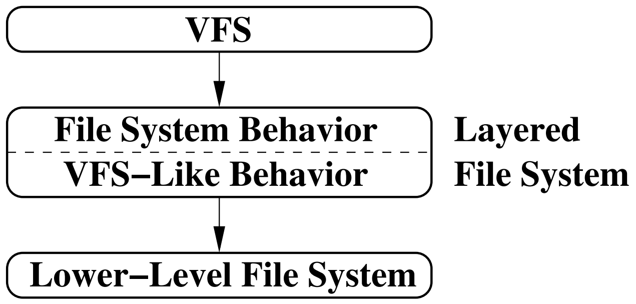

Fig. 2. On Unix, layered file systems consist of two halves: the top half must behave like a file system, and the bottom half must behave like the VFS.

lower-level objects need to be copied to the upper-level objects after each file operation. The existing VFS API requires modifications to avoid this data redundancy. In particular, the data fields should not be directly accessible from outside a file system. Instead, reads and writes of these private fields should be performed by calling file-system methods. Our evaluation, described in Appendix A, shows that the overheads of these method calls would be negligible.

FiST [Zadok 2001] is a high-level layered file system definition language that we developed. FiST includes Wrapfs [Zadok et al. 1999], a layered vnode interface implementation for Linux, FreeBSD, and Solaris. Wrapfs is a simple pass-through layer that intercepts name and data operations. The VFS objects of each entity (file or directory) are duplicated at the Wrapfs level. On Linux, many file systems have been derived from Wrapfs, including ones for encryption [Corner and Noble 2002; Halcrow 2004; Shanahan 2000], intrusion detection [Keromytis et al. 2003], integrity checking [Kashyap et al. 2004, LISA], unification [Klotzbücher 2004; Wright et al. 2006], tracing [Aranya et al. 2004], versioning [Muniswamy-Reddy et al. 2004], replication [Tzachar 2003], compression [Zadok et al. 2001; Indra Networks 2004], RAID-like functionality [Joukov et al. 2005], migration [Schaefer 2000], and more.

When Wrapfs is used on top of a disk-based or a network file system, both layers cache the pages. This is useful to implement features like encryption, where the lower level file system's page and Wrapfs's page are different (ciphertext vs. plaintext). However, cache incoherency may result if pages at different layers are modified independently. Wrapfs performs its own caching and does not explicitly touch the caches of lower layers. When writing to disk, cached pages in Wrapfs overwrite the lower level file system's pages. This policy correlates with the most common case of cache access, through the uppermost layer. To maintain coherency between the meta-data caches, Wrapfs performs a copy of the attributes from the lower level vnodes to the upper level vnodes after each operation that modifies meta-data. For example, after a file write, the new values for size, modification time, etc. are copied from the lower-level inode to the Wrapfs inode. This is required because the Linux VFS stores mutable values as fields in the inode.

3.4 Layering on Windows

Windows applications use Win32 API calls such as `ReadFile` to perform file I/O operations. As seen in Figure 3, the Win32 subsystem in turn invokes the corresponding native system calls, which are handled by the I/O manager. The I/O manager uses a message-passing architecture for all I/O-related activity in the kernel [Solomon and Russinovich 2000].

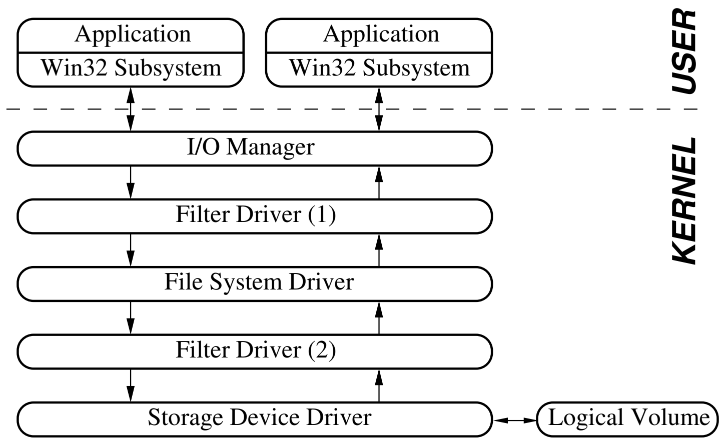

Fig. 3. Windows layering model. Applications issue system calls, which are dispatched by the I/O manager. The I/O manager routes the request via an IRP to the appropriate file system and filter drivers, and finally to the storage device driver which operates on the logical volume (partition). Filter drivers can intercept calls to file system and storage device drivers.

The majority of I/O requests are represented by a data structure called the *I/O Request Packet* (IRP) that describes an I/O request completely. The I/O manager sends the IRP from one I/O system component to another. For example, when the Win32 operation `CreateFile` is called to open a file, the I/O manager creates an IRP describing the operation and sends it to the file system driver. After the driver receives the IRP, it performs the specified operation, and passes the IRP back to the I/O manager.

IRP structure. In Windows, each logical volume has an associated driver stack, which is a set of drivers layered one on top of another. The Windows Driver Model introduced in Windows 98 and 2000 has two types of drivers: *function drivers* and *filter drivers* [Oney 2003]. A function driver is a kernel module that performs the device's intended functionality. It is analogous to a lower-level file system module or disk device driver in the Unix model. A filter driver is a kernel module that can view or modify a function driver's behavior, and can exist above other function drivers. For example, in Figure 3 Filter Driver 1 is above the file system, but Filter Driver 2 is above the storage device. Each driver owns and maintains a *device object* that represents its instance on the layered stack of drivers. A file system filter driver can view or modify the file system driver's IRPs by attaching itself to the corresponding driver stack. In a driver stack, a higher-level driver has access to the device object of the driver immediately below it. Using this object, the filter driver can send an I/O request to the lower-level driver through the I/O manager.

An IRP describes an I/O request completely. The IRP has two components. The first part is fixed: it contains fields that are common to all I/O requests, including the address of the buffer, flags, and other IRP maintenance fields. The second part is an array of `IO_STACK_LOCATION` structures with one element for each of the layered drivers (i.e., function and filter drivers), which describes the current I/O request identified by a major and minor function codes. For example, the `NtReadFile` system call creates an IRP with the major function `IRP_MJ_READ`. The `IO_STACK_LOCATION` also contains a parameters field that is specific to the request type.

Each IRP is identified by a major and minor function. The major number determines which driver function is called, and the minor number indicates the operation that the function should perform. If a filter driver does not provide a routine for a major function, neither it nor any module below it receives those IRPs. Filter drivers commonly use a single routine that is similar to the generic bypass routine in the UCLA and FreeBSD models for many major function codes. There is no default routine for unhandled IRPs, so a filter driver needs to keep up with the latest OS release to ensure proper functionality.

When an application makes a request to a device with multiple drivers on its stack, the I/O manager first sends an IRP to the highest-level driver in the stack. The request's major and minor codes, as well as arguments, are located in the `IO_STACK_LOCATION` structure corresponding to the highest-level driver. The highest-level driver does its part of the I/O processing, and then has two options. It can tell the I/O manager to complete processing of this IRP without any further processing, or it can pass the IRP down to the next lower-level driver. If the driver decides to pass the IRP to the lower-level driver, it sets up the corresponding `IO_STACK_LOCATION` structure (often by copying its own stack location into the lower-level driver's location). The higher-level driver can optionally register a completion routine that is called when the lower-level driver finishes processing this IRP. Using this callback, the higher-level driver can post-process the data returned by the lower-level driver. For example, an encryption layer registers a completion routine that decrypts the data after a read request.

Fast I/O. A significant portion of read and write operations can be satisfied from the file system cache. This fact is exploited by the Windows I/O manager by calling specialized file system routines that move data from the cache into the user's memory, or vice versa. This eliminates the need to allocate and process an IRP, which may dominate the cost of cached accesses. The file system driver exports *fast I/O* entry points that the I/O manager calls to perform the operation [Microsoft Corporation 2004b]. If a fast I/O access cannot be satisfied from the cache, then it returns a failure. On such failures, the I/O manager proceeds with the normal IRP call path.

Filter manager. Windows XP Service Pack 2 introduced a new file system filter driver architecture called the *File System Filter Manager* [Microsoft Corporation 2004a]. In this architecture, file system filters are written as *mini-filter* drivers that are managed by a Microsoft-supplied filter manager driver. Mini-filters register with the filter manager to receive only operations of interest. This means that they do not need to implement filtering code for all I/O requests, most of which would be passed along without any changes. For example, an encryption mini-filter could intercept only read and write requests. Mini-filters can also chose to ignore certain classes of I/O such as paging or cached I/O.

A mini-filter provides optional pre-operation and post-operation routines for each type of I/O request. A new structure that describes the operation is passed to these routines instead of IRPs. Mini-filter drivers can focus on request processing instead of IRP maintenance, making them easier to develop than legacy filter drivers. In the pre-operation routine, the mini-filter may complete the operation (bypassing lower layers); pass it to lower layers; or redirect the request to another instance of itself on another volume's driver stack.

Table I. A summary of the key properties of each layering model.

	Rosenthal	**UCLA**	**Solaris**	**FreeBSD**	**Linux**	**Windows IRP**	**Windows Mini-filter**
Architecture	Vnode	Vnode	Vnode	Vnode	Vnode	Message Passing	Function call
Designed for layering	Yes	Yes		Yes		Yes	Yes
Linear stacking	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Fan-in stacking		Yes	Yes	Yes	Yes		
Fan-out stacking	Yes	Yes	Yes	Yes	Yes		
Cache coherency	Yes	Yes				Yes	Yes
Attributes methods	Yes		Yes	Yes		Yes	Yes
Extensible operations		Yes		Yes			
Default operation		Yes		Yes			
Generic bypass		Yes		Yes		Yes	
Embeds functionality					Yes		

The filter manager provides several services to mini-filter drivers that ease development: context management for per-stream, per-open-file-handle, and per-driver data; a user-kernel communication channel; and temporary file name generation. However, the most important service that the filter manager provides is that it obviates the need for IRP maintenance.

In the Windows legacy model, filters cannot be unloaded, and a new filter can layer only on top of the highest driver on the driver stack. The newer filter manager can unload a mini-filter, and lets the mini-filter control its location in the stack using *altitudes*. An altitude is a fixed location in the stack (assigned by Microsoft), that identifies where in the stack a particular component is. Altitudes are designed such that a new filter can always be inserted between two existing filters. This not only provides flexibility to attach filters in the middle of the stack, but also allows them to fix their positions. For example, anti-virus filters must be above encryption filters so that the anti-virus filter can scan plain text data.

3.5 Model Summary

Table I summarizes the key properties of each layering model that we examined.

Architecture: The Rosenthal, UCLA, Solaris, FreeBSD, and Linux layering models are all vnode based. The Windows IRP model uses message passing and the Windows mini-filter model uses function calls.

Designed for layering: The Rosenthal, UCLA, FreeBSD, and both Windows models were designed with layering in mind. The Solaris and Linux models were not, and therefore it is more difficult to develop layered file systems for Solaris and Linux than for the other models.

Linear stacking: All models support linear stacking.

Fan-in stacking: The Rosenthal model does not support fan-in stacking, because the push and pop interface prevents access to the lower-level layers. Both Windows models do not support fan-in stacking for similar reasons. The UCLA, Solaris, FreeBSD, and Linux models do support fan-in stacking.

Fan-out stacking: The Rosenthal, UCLA, Solaris, FreeBSD, and Linux models all support fan-out stacking. The Windows IRP and mini-filter models do not, because they were designed specifically with linear stacking in mind.

Cache coherency: The Rosenthal and both Windows models provide cache coherency naturally, because fan-in access is not possible (i.e., one higher-level layer and another lower-level layer cannot be accessed simultaneously). The UCLA model supports fan-in access, but provides a stacking-aware cache manager that ensures cache coherency. The Solaris, FreeBSD, and Linux models do not provide cache coherency for fan-in access.

Attribute methods: The Rosenthal model replaces all vnode attributes with methods. This allows layered file systems to modify these attributes easily, and eliminates cache coherency issues. The Solaris, FreeBSD, and both Windows models also have methods for attributes, whereas UCLA and Linux do not.

Extensible operations: Only the UCLA and FreeBSD models allow file systems to add new vnode methods. The other models do not provide such extensibility.

Default operation: The UCLA and FreeBSD models allow file systems to specify a default operation. If a file system does not define an operation, then the default method is called instead.

Generic bypass: The UCLA and FreeBSD models encapsulate information in the VFS method calls so that layered file systems can use a generic routine to bypass most methods (by setting it as the default method). The Windows IRP structure provides similar encapsulation, so a single function can handle most layered operations. However, as Windows does not support a default operation, the programmer must still set each file system operation's function to this bypass function. The other models do not support a generic bypass method.

Embeds Functionality: The Linux VFS embeds a significant amount of functionality within the VFS itself. This enables rapid development of standard disk-based file systems, but greatly complicates layered file systems, which must emulate the VFS. An improved design would be to provide generic functions that can be used by the disk-based file systems. Other models do not embed functionality to such a degree as Linux.

4. CLASSES OF LAYERED FILE SYSTEMS

In this section we divide layered file systems into five broad categories, starting with the least complex file systems and moving towards the most complex ones. For each category, we describe several applications and how the different layering models affect a particular class of file systems. In Section 4.1 we describe file systems that only monitor, but do not change their operations or arguments. In Section 4.2 we describe file systems that change file data, but not operations or meta-data. In Section 4.3 we describe size-changing file systems, which modify both data and meta-data. In Section 4.4 we describe file systems that change their operations. In Section 4.5 we describe fan-out file systems, which layer on several underlying file systems.

4.1 Monitoring

Monitoring file systems intercept operations and examine their arguments, but do not modify them. Examples of monitoring layers are tracing, intrusion detection systems (IDSs), and anti-virus. Monitoring file systems use linear layering as shown in Figure 1(a). Fan-in access as shown in Figure 1(c) is inappropriate because accesses should not bypass the monitoring layer.

Both tracing and IDSs need to examine a large subset of file system operations. For a tracing system, the operations' arguments are important to gain insight into application behavior. For an IDS, the arguments and processes are needed to determine which operations should be considered a sequence; and the arguments are needed to correlate one operation with other operations [Hofmeyr et al. 1998]. When tracing is used to gain insight into systems, performance is critical; the layered file system should be able to log the operations and timings.

The Rosenthal, UCLA, and Unix models are all suitable for tracing to varying degrees. In all three of these models, simple linear layering can be used. Because fan-in access is not required, the Rosenthal and Unix models do not suffer from cache coherency issues. The Rosenthal model has fixed operations vectors, which means that for simple pass-through layering, each operation requires its own function. On the other hand, as all vnode fields are accessed through methods in the Rosenthal model, no vnode access information is lost. The Solaris model is similar. Because the UCLA model it is extensible, the vast majority of operations can be traced with only a single generic routine. The UCLA model passes meta-data to the operation, so operation names (or unique identifiers) and vnodes can be logged. On the other hand, vnode fields may still be accessed without the tracing layer's knowledge. The Windows model is similar to the FreeBSD and UCLA models in that a single routine can generically pass an IRP to the layer below. It does not expose information in the vnode fields (on Windows the fields are stored in a file control block), so the tracing system has full access.

The Linux VFS makes tracing more difficult. Fields are directly accessed by the VFS, without method invocations. The Linux VFS interface is also asymmetric. There are several maintenance routines (e.g., removing a tree rooted at a particular dentry from the name space) that the VFS calls, which cannot be passed to the lower-level file system. After certain events occur (e.g., reference counts reach zero), the VFS calls these methods and they cannot be intercepted by the tracing layer. The FreeBSD VFS provides the best support for tracing. The vnode fields accessed only through method invocation, and it is possible to use a generic tracing routine, because FreeBSD packs arguments and meta-data about those arguments into a structure.

Anti-virus file systems need to examine only a small subset of operations. For example, many commercial virus scanners only intercept `exec` and either `open` or `close` [Network Associates Technology, Inc. 2004; Sophos 2004; Symantec 2004]. Other virus scanning layers intercept `read` and `write`. The UCLA, FreeBSD, and Windows models make this trivial because the rest of the operations can be serviced by a generic bypass routine. The Rosenthal, Linux, and Solaris models require layered implementations for each routine.

Anti-virus systems must be able to access the file data. When using an `open`, `close`, or `exec` method for virus scanning, this requirement boils down to the ability to read an open file from the kernel. Each model easily provides this functionality. In the Rosenthal, UCLA, and Unix models, the anti-virus layer can make simple VFS function calls. In the Windows model, instead of using function calls, IRPs are generated within the layer's completion routine for the `open` or `close`. Unfortunately, generating these IRPs is difficult and error-prone. Using the new filter manager architecture, it is possible to create a brand new file handle that directly accesses the lower-level layer using the newly introduced filter manager calls that are similar to the standard in-kernel file-access API.

When using `read` and `write`, things become more complex, particularly for `mmap` operations where all data-oriented operations need to be intercepted. The Rosenthal model provides for intercepting page data by enforcing the rule that only the top-level file system may cache pages, and all page-based operations go through the top-level object. In the UCLA model, file systems can use *page flipping* to efficiently move pages that do not change between layers. In Linux, each memory region in a process, called an *address space*, has its own operations vector and *host* (i.e., the inode or other object that the region represents). Each page belongs to precisely one address space. The page structure points to its address space's operations vector. This means that to intercept `mmap` operations, a layered file system must maintain its own pages. Using traditional layering, this effectively halves the cache size. Recently, we have employed a method similar to UCLA's *page flipping* [Joukov et al. 2005]. Wrapfs allocates its own page, but then replaces its own address space's host with that of the lower-level layer. The lower layer then fills in Wrapfs's page without the need to allocate a separate page. In FreeBSD and Solaris, page-based operations are part of the vnode; therefore, interception without double caching is simpler. Windows uses a single path for `read` and `write` system calls and `mmap`; therefore, intercepting the read and write IRPs is sufficient.

Many monitoring file systems require complex algorithms. For example, an anti-virus algorithm scans data for signatures or uses other complex heuristics. It is often preferable to run complex portions of code in user-space, but the rest of the file system in the kernel (for improved performance). We have used several methods to enable this communication on Linux. In our Elastic Quota file system, we needed to store a persistent mapping from inode numbers to names, to locate all instances of a given file. We chose to use the BDB database [Seltzer and Yigit 1991], but at the time it only ran in user-space. We used a Linux-specific *netlink* socket to send messages from the kernel to a user-level program, which manipulates the BDB database accordingly. The Windows Filter manager supports an abstraction called a *communication port* that is similar to a netlink socket. We also used another more generic option: a character device driver. The kernel can enqueue messages that are returned on `read` and execute functions during the `write` handler. As character devices are available on most Unix systems, this approach is portable in that the same user-level program can run on several OSes. FUSE (Filesystem in Userspace) uses a similar approach [Szeredi 2005].

4.2 Data Transforming

One of the most common types of a data-transforming layer that does not change the data's size is encryption [Corner and Noble 2002; Halcrow 2004; Wright et al. 2003] (for size-changing transformations, see Section 4.3). Data transformations have similar requirements to monitoring `read` and `write`, but for efficient backup they should additionally support fan-in access as shown in Figure 1(c). Fan-in access results when a layered file system is mounted on one path, but the underlying data is still available through a separate path. For example, when using a compression layer, user applications must go through the compression layer so that they can process uncompressed data. For performance, a backup system should bypass the compression layer and write only the compressed data to tape.

Rosenthal's model, UCLA's model, Linux, and Windows easily support modifying the data on the `read` and `write` paths (or equivalent page-based ones). Neither FreeBSD nor Solaris include layered file systems that modify the data, but the FiST templates for FreeBSD and Solaris do support data modification.

Rosenthal's model is not compatible with fan-in, as lower-level vnodes cannot be directly accessed, because each vnode is essentially an alias for the top-level vnode. Therefore it should not be used for data-transforming file systems. The Windows model suffers from a similar disadvantage: all accesses are routed through the highest-level filter driver, so it also cannot support fan-in access. UCLA's cache coherence model supported fan-in layering. The FiST templates support fan-in access, but without the cache coherence provided by the UCLA model: if a page is independently modified in the higher-level and lower-level layers, then updates may be lost.

On Linux, attributes (e.g., size or permissions) may also be out of sync between the higher-level and lower-level layers; this is because the attributes are copied only after certain operations that write the data. This means that if lower-level attributes are modified, the higher-level layer does not reflect those changes until the file is modified through the higher-level layer. On FreeBSD and Solaris, this is not an issue because the vnode does not contain attributes but instead uses a `getattr` method to retrieve the attributes.

Sparse files contain regions that have never been written, called *holes*. When data from a hole is read, the file system returns zeros. For data transforming file systems (e.g., encryption), it is possible for zeros to be a valid encoding (e.g., ciphertext) [Halcrow 2004; Zadok and Nieh 2000]. This means that if a data-transforming file system reads zeros from the lower-level file system, then it decodes them and returns incorrect data. In FreeBSD, a vnode operation called `bmap` maps a file's logical block number to a physical block. If the logical block is a hole, then –1 is returned. Using `bmap`, a layered file system can determine if a page is indeed a hole, or actually filled with zeros. Linux also has a `bmap` vnode operation, but it is deprecated. Solaris's UFS uses a similar function, but it is not exported by the VFS. Unfortunately, the `bmap` function is not always defined (e.g., NFS has no `bmap` operation), therefore `bmap` is not a portable method of determining whether a given page is a hole. To solve this problem, data-transforming file systems often prevent sparse files from being written. If a write takes place past the known end of the file, then zeroes are transformed (encrypted) and the transformed data is written from the current end of the file to the point where the write after the end of the file occurs (i.e., the holes are filled in). Unfortunately, filling in these holes can markedly reduce performance for some applications that rely on sparse files (e.g., certain databases).

4.3 Size Changing

Some transformations of file data result in changes to its size. A simple example is compression: layered compression file systems reduce the overall size, which saves storage space and transmission bandwidth. Such file systems require transformation algorithms that take a certain number of bits as input and produce a different number of bits of output. We refer to these algorithms as *Size Changing Algorithms* (SCAs) [Zadok et al. 2001]. Most SCAs involve whole streams of input data, such that the reverse transformation requires the entire output. For example, if compressing 4,096 bytes produces 3,000 bytes of output, then those exact same 3,000 bytes need to be decompressed to reconstruct the original data.

Several SCAs have been implemented as extensions to existing disk-based file systems. Windows NT supports compression on NTFS [Nagar 1997] using a filter driver. E2compr [Ayers 1997] is a set of patches to Ext2 that adds block level compression. The disadvantages of these methods are that they work only with specific OSes and file systems. A file system layer that implements an SCA can be used with any existing file system.

Supporting SCAs at the layered level is complicated because they change the file size and layout, thereby requiring maintenance of additional mapping information between the offsets for each file in the different layers of the file system. If not designed carefully, this mapping information could cause additional processing and I/O, significantly affecting performance. For example, if an encryption file system encrypts one page at a time, but the encryption algorithm produces more than one page of output, then three challenges arise. First, a single page at the encryption level represents more than one page in the lower level file system. Therefore a read request for one page results in two pages of I/O at the lower-level, increasing the possibility for bugs and poor performance. Second, the precise offset to read needs to be determined by the offset mapping information, because different pages may have different output sizes. Third, when retrieving file attributes (e.g., using `stat`), the encryption file system should display the logical size at the higher-level, not the size from the lower layer. This means that each `getattr` operation (or `lookup` on Linux) needs to access the mapping information to find out what the decrypted size is. Therefore, it is crucial to access offset mapping information persistently and efficiently.

Fast indexing [Zadok et al. 2001] provides SCA support in layered file systems. It uses estimation heuristics and optimizes performance for the more common and important operations (e.g., reads or appends), while rarely executed operations (e.g., overwriting data in the middle of the file) may incur greater overheads. Fast indexing stores each file's SCA-mapping information in an *index file*. This file serves as a fast meta-data index into the encoded file. The fast indexing system encodes and decodes one or more pages at a time. The offsets of the pages within the encoded file are stored as a table in the index file. When reading the file, the index file is consulted to determine the size of the decoded file and the offsets to read within the encoded file.

4.4 Operation Transformation

Many features can be implemented by transforming data or meta-data, but certain advanced features like versioning require transforming the operations themselves. For example, to maintain versions of files at the file system level, operations like `write`, `unlink`, etc. must be transformed into a copy or a `rename` at the layered level. While transforming operations, it is important that the semantics of the OS are maintained. For example, if `unlink` is called in Unix and it returns successfully, then the object should not be accessible by that name any more. One way to maintain the OS semantics while not exactly performing the operation intended, is to transform one operation into a combination of operations. For example, to preserve the data of a file after it is unlinked, the unlink operation could be transformed to a copy and then invoke an `unlink`. In Windows, operation transforming is easier because there is no notion of a vnode structure. Instead, each operation takes the full path name (or open file handle) of an object. The path name can then be changed using simple string manipulation before it is passed down to the lower level.

To transform operations, the argument set of one operation must be transformed into the argument set of the new operation. The relative ease of these transformations depends on the implementation of the vnode interface. In Linux, the dentry cache allows an object's parent to be referenced, which means that one can traverse the dentry structure until the root dentry is reached. This can be used to reconstruct paths, or to operate on an object's parent (e.g., to manipulate objects with `rename` and `unlink`, you need to access the parent directory). FreeBSD and Solaris do not have a parent pointer from their vnode objects, which can make these operations more difficult. For example, in FreeBSD union mounts, the vnode private data includes a pointer to the parent vnode, to simplify the `lookup` of "`..`" [Pendry and McKusick 1995].

Having many different types of VFS objects in Linux (`file`, `dentry`, `inode`, etc.) makes operation transformation at the layered level more difficult. Many operations take different kinds of objects as arguments, which necessitates the conversion of one type of object to another when transforming the operation. Some of these conversions are even impossible. For example, in Linux, it is not possible to transform a memory-mapped write into a copy, as a memory-mapped writes take the `inode` object as argument, whereas copying requires the `dentry` object to open a file, and there is no support for converting inodes to dentries in the Linux vnode interface. This scenario occurs in versioning file systems that needs to perform copy-on-write for creating new versions [Muniswamy-Reddy et al. 2004]. In FreeBSD and Solaris, almost all operations take a `vnode` as an argument, so transforming the operations is simpler. Similarly, in Windows, all operations are implemented in terms of path names and file objects, so converting between objects is simpler.

4.5 Fan-Out File Systems

A fan-out file system layers on top of several lower-level file systems called *branches*, as shown in Figure 1(b). An object in a fan-out file system represents objects on several lower-level file systems. Fan-out file systems still need to deal with all of the previously mentioned issues and two more key challenges. First, each operation on the fan-out level may result in multiple lower-level operations. This makes guaranteeing each operation's atomicity difficult. Second, the VFS often uses fixed-length fields that must be unique. This complicates mapping between upper and lower level VFS objects and fields.

Each VFS operation should be atomic, but an operation on a fan-out file system may involve several underlying branches. For example, to remove a file in a fan-out mirroring file system, it must be removed from each underlying branch. If the removal fails on any branch, then this partial error needs to be handled. Rosenthal proposed that transactions would be necessary to handle these conditions [Rosenthal 1992], but did not implement them. Adding transaction support to the OS would require significant and pervasive changes to many subsystems, not just the VFS. For example, process control blocks contain pointers to open files. If a transaction that created a file were to be aborted, then those pointers need to be removed from the process control block. Instead, fan-out file systems have implemented their own specific policies to minimize the non-atomicity of their operations. For example, UCLA's Ficus system is designed to support disconnected replicas and resolve conflicts [Guy et al. 1990]. FreeBSD's union mounts avoid partial failures by allowing writes only to one branch [Pendry and McKusick 1995]. In our unification file system, we have carefully ordered operations such that we defer changing the user-visible union until the very last operation [Wright et al. 2006]. If the operation succeeds, then the whole operation returns success and the union is modified. If the last operation fails, then the underlying branches may have changed, but the overall view remains the same. Transactions are gaining more acceptance as an OS primitive [MacDonald et al. 2002; Microsoft Corporation 2004c], but even if fully integrated into an OS there are still cases in which transactions will not be sufficient. For example, if one branch is located on an NFS file system, then transactions cannot be used reliably because NFS does not support transactions.

Many VFS fields have a fixed length that cannot be extended. This is true of the Rosenthal, UCLA, Solaris, FreeBSD, and Linux models. For example, inode numbers are commonly 32-bit integers. They are used by user-space processes and other kernel components to identify a file uniquely (e.g., `tar` only writes one copy if two files have the same inode number). Any file system may use this entire address space; so if two file systems are combined, then there is no direct way to compute a new unique inode number based on the existing inode numbers because there are no remaining bits to distinguish between the two file systems. Directory offsets record at what point a process is when reading a directory. In a traditional directory, the offset is a physical byte-location in a file, but in more modern directory structures, such as HTrees [Phillips 2001], the offset is actually a small cookie. Using this cookie, the file system can resume reading a directory from where it left off. A fan-out file system cannot use any additional bits to record in which branch to resume the directory-reading operation. Both of these problems could be alleviated if instead of allocating fixed-size units for identifiers, the VFS supported variable-sized arguments. A fan-out file system that needs extra bits could simply concatenate its own information with that of the lower-level file system.

The Windows model has fan-out problems because it was designed for linear layering. To support fan out, a filter driver must be interposed on each branch's driver stack. When an I/O request is received on any stack, it must be sent to the lower-level driver on each driver stack. Because the filter driver is interposed on each volume's driver stack, each of the volumes becomes an alias for the fan-out file system. Because Windows does not support fan in, each of the lower-level volumes is now inaccessible; this may limit some applications of fan-in file systems (e.g., when running a snapshotting file system, programs and another instance of the filter driver could not access older snapshots).

When using the Windows filter manager, creating instances of a mini-filter is controlled by the filter manager. A single mini-filter instance cannot be interposed on multiple volumes, so a separate instance is needed for each volume stack. This makes fan out more difficult to implement with the filter manager, but the IRP model is still available. To support fan out, a mini-filter can duplicate the I/O request and send it to the lower-level layer. The original request can then be redirected to another instance of the driver on a different volume's driver stack. Having two separate instances of the driver complicates development of fan-out file systems on Windows more so than Unix.

5. USEFUL VFS AND OS FEATURES

We developed over twenty layered file systems on a range of OSes for more than a decade. During that time, we have repeatedly come across several deficiencies in file system and OS infrastructures. In this section we categorize these limitations into seven general problems (the first five of which we have solved or worked around). With each category, we also discuss enhancements to existing file system and OS infrastructures that would aid in easier development and improved performance of layered file systems. In Section 5.1 we discuss persistently storing extra meta-data. In Section 5.2 we discuss missing interception points. In Section 5.3 we discuss access control. In Section 5.4 we discuss cache management. In Section 5.5 we discuss collapsing layers. In Section 5.6 we discuss the last two issues: intents and file serialization.

5.1 Persistently Storing Data

Layered file systems often need to store information persistently on disk for various scopes: per–file-system, per-directory, or per-file. For example, a simple integrity-checking file system that performs checksumming on file data needs to store the checksums in a persistent manner. An encryption file system needs to store the encryption algorithm and parameters on a per-mount basis. The same encryption file system may store security policies for file trees on a per-directory basis. Layered file systems most commonly use per-file data. For example, file system layers that perform size-changing transformations persistently store the mapping information for each file. Often, per-file data is accessed during critical regions of `read` or `write`. For example, a checksumming file system accesses the checksum associated with a file (or page) during every read and write. Therefore, persistent data must be accessed efficiently. The persistent data storage method adopted, in addition to being efficient, should also be easy to implement, and transparent to the other layers. In the rest of this section we describe some methods we have used to store persistent data, and suggest new OS facilities to improve persistent storage by layered file systems.

Unused fields and bits. Some lower-level file systems (e.g., Ext2) have unused inode bits or fields that layered file systems can take advantage of. These flags can be used by layered file systems in a transparent fashion to store file-specific information. Elastic Quotas [Zadok et al. 2004] use the `no dump` bit in Ext2 inodes to differentiate between two classes of files. While this method has good performance, the layered file system must be tailored to a specific lower-level file system. This violates the general design goal of layering: to support a wide range of lower level-file systems transparently; but performance benefits sometimes justify its use.

Parallel files. The parallel file method stores per-file data in a separate file F' for each file F [Muniswamy-Reddy et al. 2004; Zadok et al. 2001]. Parallel files are used in our compression file system for size-mapping information and in our versioning file system for per-file policies. Parallel files are easy to implement; however, we found three disadvantages to this approach. First, having two physical files for each regular file in a file system requires double the number of in-memory objects, on-disk inodes, and directory entries. Second, a good naming scheme for parallel files is required, but even with the best naming scheme, a user might want to create a regular file with the same name as the parallel file. Because the name of the parallel file has to be larger than the actual filename for correctness, it shortens the effective `MAX_PATH` limit. Third, journaling file systems provide only per-file consistency. Because the consistency semantics of parallel files span more than one file, journaling cannot provide consistency between F and F'.

Interspersed data. Another option is to embed persistent data in the main file, so that extra objects are not needed. Efficiency and access patterns often dictate the data embedding method: at the end of the file, at the beginning of the file, or spread out throughout the file. Placing persistent data at the end of the file is useful for append-mostly access patterns. However, that data should be small (e.g., an integer) to avoid having to rewrite too much data at the end of that file. Placing persistent data at the beginning of the file is useful for sequential read patterns, because the extra data can be read before the main file is read. However, if the size of the data is not a multiple of the page size, this complicates the computation of offsets into the main file's data. The most flexible technique is to spread the data evenly throughout the file and to align it on page boundaries. The two advantages here are that meta-data pages could be easily computed and meta-data can be efficiently read separately from the data pages of the main file. The main disadvantage is that this persistent meta-data should fill most of the extra pages, so space is not wasted. We used this technique twice. Our encryption file system stores integrity checksums throughout the file, and our RAID-like layered file system (RAIF [Joukov et al. 2005]) stores parity information, stripe size, and more.

For example, in our encryption file system, with a 160-bit SHA1 hash, 204 hashes fit within a single 4KB page, so every 205th page is a checksum page. With this method, we can easily compute the original file size and the index of any given page. Because each upper-level page corresponds to precisely one lower-level page, there is no unnecessary I/O. The main disadvantages with this method is that to access any one of the 204 pages requires access to the checksum page, and that the OS may turn off read-ahead because file access is no longer sequential.

In-kernel databases. KBDB [Kashyap et al. 2004, TR] is our port of the Berkeley DB [Seltzer and Yigit 1991] to the Linux Kernel. Berkeley DB is a scalable, transaction-protected data management system that stores key-value pairs. I^{3}FS, our layered integrity-checking file system, uses in-kernel databases to store the policies and checksums associated with each file [Kashyap et al. 2004, LISA]. KBDB has two advantages for persistent storage: (1) it provides a common and easy-to-use interface for complex data structures like B-trees and hash tables, and (2) it is portable across systems as data can be backed up by simply copying the database files. The main disadvantage of using KBDB is that to achieve good performance, the schema must be carefully designed and the database parameters must be tuned through experimentation.

Extended Attributes. Extended Attributes (EA) associate arbitrary name-value pairs with each file. They can be used to store information like ACLs or arbitrary user objects. On Linux, Ext2, XFS, and Reiserfs support extended attributes using a common API. FreeBSD's FFS also supports extended attributes, but with a different API. Layered file systems can use the EA interface to store persistent per-file data. However, there are two key disadvantages of using EAs. First, each file has a limited space for EAs (e.g., 4,096 bytes on Ext2). Second, each file system's EA implementation has distinct performance and storage properties. We have developed a generic EA file system layer (called EAFS) that provides EA support for any existing file system with no practical limits to the number or size of EAs.

Data streams. NTFS supports multiple named data streams for a file. Each stream can be opened and accessed as if it were a separate file. When mounted over NTFS, layered file systems can use streams to store persistent data associated with files, but streams cannot be transparently used on different file systems. Streams are also unavailable on Unix.

New OS features. Clearly, each of the methods we have adopted for storing data persistently have their own advantages and trade-offs in terms of performance, portability, and extensibility. Lower-level file systems should provide more efficient and extensible persistent meta-data storage for other OS components, including layered file systems.

The meta-data management mechanism of on-disk file systems should be made extensible. Extended attributes should be generalized across file systems, without size limitations. For example, a reasonably sized ACL and per-user encryption keys for a layered encryption file system may not fit within a single 4KB block. EAs should also have predictable performance requirements across lower-level file systems (e.g., some file systems perform well with many duplicate EAs, but are poor for searching; others have efficient search but have poor duplicate storage). Our EAFS provides such predictable and uniform functionality. File systems should also support additional named streams (as in NTFS), using a common API. This is important as EAs are inappropriate for arbitrarily large data (e.g., per-block information should not be stored in EAs because you cannot efficiently `seek` to a specific position within an EA).

Current file systems only support three operations on file data: read, overwrite, and append. Two new operations should be created: *insert page* to insert a page in the middle of a file, and *delete page* to efficiently delete a page from the middle of a file. This method would avoid unnecessary movement of data (e.g., to insert a page, the remaining data currently needs to be rewritten). This feature can also be extended to user level so that applications can efficiently manipulate data in the middle of files (e.g., for text or video editors).

Current on-disk file systems perform block allocation by themselves and they do not export interfaces to the VFS for advisory allocation requests. PLACE [Nugent et al. 2003] exploits gray-box knowledge of FFS layout policies to let users place files and directories into specific and localized portions of disk, but requires a separate user-level library and cannot always place data efficiently as it must work around the file system's allocation policy. If the VFS can give hints about related data to lower level file systems, then it can be used in the block allocator to ensure that related information is placed as close as possible. Currently, none of the OSes we reviewed support this feature; but this would be useful when a layered file system uses parallel files.

5.2 Missing Interception Points

One of the most frustrating, yet all too common, aspects of developing layered file systems is that some operations cannot be intercepted. For example, the `chdir` system call on FreeBSD, Solaris, Linux, and Windows does not call any file system methods except `lookup`, `getattr`, or `access`. This causes problems with our unification file system (Unionfs), which supports dynamic addition and removal of branches. When a branch is added, it is analogous to mounting a file system. Unionfs must keep track of how many referenced files there are on each branch to prevent an in-use branch from being removed. Because `chdir` is not passed to the file system, the unification file system is unaware that a process is using a given branch as its working directory. When the branch is then removed, the working directory of the process no longer exists. In Unionfs, when the directory is next accessed, we replace its operations vector with one that returns `ESTALE`, which is used by NFS clients when an object is deleted on the server. This solution is hardly ideal; the VFS instead should notify file systems when a process changes its working directory. Indeed, *every* file-system–related system call should have a corresponding vnode operation.

Another example of a missing interception point is process termination. All vnode operations are executed in the context of a process [Kleiman 1986], yet when the process terminates, the file system is not notified. We came across this issue when developing our layered encryption file system [Wright et al. 2003]. Instead of granting access to individual users, our file system could specify more fine-grained access control (e.g., by session ID or PID). To authenticate, a process executes an `ioctl` and passes the authentication information to the encryption file system. The encryption file system makes note that this process has authenticated, and then allows it to execute other operations. To do this securely, we need to invalidate the authentication information once the process exits so that a newly created process with the same identifier could not hijack the session (we also expunge cleartext data from the cache). Other process operations including `fork` and `exec` may also be of interest to file systems. We modified the OS to provide lightweight hooks so that file systems (and other kernel components) can easily intercept the subset of these operations that are required. We have since used them for several projects such as Elastic Quotas [Zadok et al. 2004] and others [Kashyap et al. 2004, TR].

As we discussed in Section 3.3, intercepting of accesses to file system objects would allow us to deal with redundant data and meta-data efficiently (i.e., cache coherency between layered and lower-level file systems). We show in Appendix A that the corresponding API would add negligible overheads.

5.3 Access Control

Layering is a form of wrapping, and wrappers are a time-honored software security technique. Not surprisingly, layered file system have been particularly useful for information-assurance purposes. Many file systems need to modify access control decisions made by the lower-level file system. For example, our layered file system that provides ACL support for any existing file system needs to determine authoritatively what operations should be permitted.

Unfortunately, the Solaris, FreeBSD, and Linux VFS operations directly call the lower-level inode's `access` operation (`permission` is equivalent to `access` on Linux). Therefore, a layered file system can only provide more restrictive permissions than the lower-level layer without changing the effective UID of the process. To grant more privileges than the lower-level file system would otherwise grant, a layered file system must change the process's UID to the owner of the inode, perform the operation, and then restore the UID. We adopted this solution to support ad-hoc groups in our layered encryption file system [Wright et al. 2003] and in our generic ACL layer. An additional complication added by FreeBSD is that the `access` method is called from within `lookup`, before the precise operation to be conducted is known.

The Windows model delegates security entirely to the file system. When a file is opened, an `IRP_MJ_CREATE` message is sent to the file system along with an `ACCESS_STATE` structure that describes the object's context and the desired access type. If the file system decides to deny the access, then it returns an error; otherwise it performs the open. This allows lower-level file systems to implement security primitives flexibly (e.g., FAT has no security primitives, but NTFS has ACLs). However, this model makes it difficult to layer on top of existing file systems as the security information is not exposed in a uniform manner to components above the lower-level file system. The Windows kernel supports *impersonation* of a saved security context (i.e., the privileges given to a user). Unfortunately, the Windows kernel does not provide a method to change the IRP's security context. At the point that the filter driver is invoked, even if the filter driver changes the thread's security context by impersonating another user, the IRP's security context has already been set, so the impersonation has no effect for this IRP. If the impersonation is not relinquished, the thread would have the increased privileges for all future I/O operations (even for *other* file systems); therefore using impersonation is not suitable for elevating privileges precisely.

5.4 Cache Management

In modern OSes, file systems are intimately connected with the VM system and the caches. The page cache is generally thought of as residing "on the side"—that is, it does not actually exist above the file system, yet the page cache is not below the file system either. In Linux, Solaris, FreeBSD, and Windows, vnode operations (or I/O requests) exist for reading pages into the cache and writing pages back to disk. Once a page is read into the cache, the file system has little control over it. The OS often directly accesses cached pages without notifying the file system. The OS itself decides which pages to evict without notifying the file system, and file systems cannot easily evict their own pages.

Accessing pages in the cache without notifying the file system prevents the file system from applying its access-control policies to the page. The OS uses generic algorithms for deciding which pages to evict, and then proceeds to evict them without notifying the file system. In a layered file system, several objects may be related and if one is removed from the cache, then the other should be removed as well. For example, in a file system that uses parallel files for persistent data storage, if the main file is removed, the parallel file should be removed as well. However, without notification, the layered file system cannot evict those other objects. When pages must be evicted, the OS should provide a reasonable default policy, but allow file systems to override it. For example, a replication file system should evict pages from all but a single replica so that most reads can still be satisfied.

UCLA's layering model provided a centralized caching manager with appropriate notifications to the file system. Unfortunately, it required changes to all file systems. We have developed a more limited cache coherence mechanism for Linux that does not require changing existing file systems. To determine the layers' order, we modified the VFS to record information about pages as they are created. If an incoherence is detected (based on the paging hardware's dirty bits), then we handle it in two ways. First, we invalidate any stale higher-level pages that are not dirty. Second, if multiple levels are dirty, then we write-through the highest-level dirty pages. This second policy has the effect of treating the higher-level layers as authoritative. We also provide file systems with more control over their cached objects. When developing our layered encryption file system, we added kernel call-back hooks which allowed us to evict cleartext objects from the cache when a session ends [Wright et al. 2003].

5.5 Layer Collapsing

Each layered file system usually provides specific functionality, so that they can be composed together in new ways. Part of the allure of layered file systems is that they perform only one function, so that they can be composed together in new ways to provide new functionality. A common example is that a layered encryption file system can be made more secure by adding integrity checking. One way to achieve this is to mount one layered file system on top of another. When there are multiple layered file systems, existing performance issues in layering become more acute. If a layered file system duplicates file data pages and vnode objects, then an n-level mount could result in the data pages and objects being duplicated n times, and n times as high function call overheads. Care must also be taken to ensure the two file systems can be composed together without introducing incompatibilities. The transmutation operations performed by one layered file system may not be compatible with those of another (e.g., a versioning file system may convert an `unlink` to a `rename`, but that could prevent an IDS file system below it from detecting malicious file-deletion access patterns). Finally, as the number of layers increases, the overhead due to function call and object indirection becomes more pronounced.

One solution to this is to collapse several layers into a single layer during compilation or at runtime. High-level layered file system specification languages such as FiST offer a syntax for describing file systems [Zadok and Nieh 2000]. To collapse different layers during compilation, their high level specifications can be combined automatically. In the example of the encryption file system described here, the decision whether to encrypt the data first or to compute the checksums first should be specified. This method would provide seamless integration of several features in a single layer with low overheads. Even in this method, the specifications should ensure that conflicts do not arise between different layers while storing persistent data, or while using in-memory fields. Layers can be collapsed at runtime by modifying the VFS to order the operations performed by several layers so as to eliminate the multiple function calls and object references. Runtime collapsing is more flexible, but is significantly more complicated to implement.

5.6 Other Desirable File System Features

There are often cases where a single system call or operation results in several vnode operations. For example, FreeBSD requires three vnode operations to open a file: `lookup`, `access`, and `open`. The Linux and FreeBSD VFSs pass an *intent* to the earlier operations within a multi-operation sequence, so that they can modify their behavior based on the forthcoming operation [Cluster File Systems, Inc. 2002]. Right now, intents are rather limited. In FreeBSD, the `lookup` operation has lookup, create, delete, and rename intents. On Linux, there are lookup, create, and access intents. Intents should be more descriptive. For example, when executing `mkdir` on Linux and FreeBSD, an "mkdir" intent should be used instead of a "create" intent. With more specific intents, layers can be more versatile (e.g., FreeBSD's MAC and Linux's LSM file-system–related hooks could be implemented as a file-system layer). On Solaris, intents are not available. On Windows, each operation is self-contained, so there is no need for intents.

Most files, in addition to data, also have simple meta-data like access times, modification time, inode change time, an owner, and access permissions. Recently, the number of additional types of information associated with a file has been multiplying (e.g., EAs, streams, etc.). This means that it is not possible simply to copy a file on a file system to preserve its full contents, without the copy application understanding each of these specialized types of data. Layered file systems suffer from a similar problem: they cannot reliably copy a file when required. To solve this problem, each file system should provide two new vnode methods: (1) a serialization method that consolidates all extra, file-system–specific information about a given file in a series of opaque bytes, and (2) a corresponding deserialization method that restores a file based on those opaque bytes.

We plan to implement these two features in the future.

In sum, we enumerated seven problems that we encountered, five of which we provided some solution for. In light of this, we have suggested seven possible changes to OSes and VFSs that would enable more versatile, efficient, and robust layered file systems:

(1) To support persistent storage, an extensible and uniform meta-data interface and a file insert-delete interface should be implemented in lower-level file systems.
(2) OSes should notify the file system about all file-system–related operations, by identifying and adding missing interception points.
(3) Policy decisions, like permission checks, should be designed such that all layers are traversed, to allow each layer to make the appropriate security decisions.
(4) The OS should include a light-weight cache manager, like the one we built for Linux,

that provides cache coherence without requiring changes to lower-level file systems. OSes should also include hooks for a file system to have more control over its cached objects.

(5) Layer collapsing should be employed to improve performance when several layers are used together.

(6) The VFS interface should include precise intents so that file systems can accurately determine a file system operation's context.

(7) OSes should include a VFS-level interface to copy a file reliably.

Each of these seven features would provide an improved interface for future layered file system developers.

6. SUMMARY AND CONCLUSIONS

In this paper we have three contributions. First, we have analyzed four different models of layered file system development: the Rosenthal model, the UCLA model, layered development on current Unix OSes, and Microsoft Windows. Second, based on our experience developing over twenty layered file systems on four different OSes (Solaris, FreeBSD, Linux, and Windows), we have classified layered file systems into five categories, based on their interactions with the VFS and the lower-level file systems. We analyzed the OS and VFS features that each class of file systems requires. Third, we identified seven types of problems that affect all classes of layered file systems, and described our current solutions for five of them and suggested how future OSes could better address these deficiencies.

The layering model used by current Unix systems is based on Kleiman's vnode interface and ideas explored by the Rosenthal and UCLA models. To be useful, the vnode interface and the operations vector should be as extensible as possible. The UCLA and FreeBSD interfaces provide a flexible interface, with extensible argument lists and operations. The Linux VFS provides many default routines and default functionality for undefined operations within the VFS itself. This enables the rapid development of lower-level file systems, but including file-system functionality in the VFS itself makes it more difficult to implement layered file systems. Rather than including default functionality within VFS methods, generic functions that a file system can use should be provided. Solaris and FreeBSD only have one core VFS object, the vnode. This makes it simpler to transform one type of operation to another, as objects do not need to be converted from one type to another (especially because not all conversions are possible).

Windows uses a message-passing architecture for layering that was designed only for linear layering. We found that the Windows message-passing architecture and the Unix object-oriented architecture are functionally equivalent. However, other factors are more likely to affect whether a given layered file system is relatively simple or difficult to develop. Message passing may, however, be a slightly less intuitive programming model than a procedural one—as evidenced by the fact that Microsoft introduced the filter manager API, which eliminates the need for programming using a message passing API. Windows makes it relatively simple to perform monitoring, data transformation, and other linear transforms. Transforming operations is simple because Windows wholly encapsulates each operation inside an IRP and uses a few objects. Significant portions of meta-data that are exposed to higher-levels in Unix (e.g., security information) are not exposed on Windows. Therefore, it is more difficult to modify this type of information. Finally, fan-out and fan-in file systems are more difficult to implement on Windows.

Provisioning for Extensibility. In the remainder of this section, we describe five forward-looking design principles (in bold font) for OSes to support layering well. In general, it is difficult to provision for extensibility fully, because at the time you are designing an extensible operating system you do not know what types of extensions future developers will envision. Because of this, we believe that:

(i) To support layering properly, the operating system should provision for extensibility at every level.

In particular, methods, return codes, in-memory objects, and on-disk objects should all be extensible.

Methods. Operations vectors should be extensible as in the UCLA and BSD models. New operations should be easy to add, and enough meta-data should be available to bypass any operation. This allows simple layers to be developed quickly, and allows the VFS to be extended without the need to change existing file systems.

Return codes. The VFS should be structured in a way that unknown return codes are passed back to user-space. In this way, file systems can introduce new more descriptive error codes. The VFS should also provide a method to register these new error codes, an associated descriptive messages, and an associated legacy error code to map to. For example, an Anti-virus file system could return EQUARANTINED, with an associated text of "This file may contain a virus and has been quarantined," and a legacy-compatibility error code of EPERM. In this way enhanced applications can present more informative messages to users, while existing applications work without any changes.

In-Memory Objects. The VFS should not directly access fields of objects but wrap these accesses in methods instead. As demonstrated in Appendix A, this exacts only a negligible overhead. Additionally, each in-memory object should have a private data field that can be used to extend the object (e.g., to add a pointer to a lower-level object).

On-Disk Objects. File systems should provide a way to store arbitrary meta-data with each object. All non-standard on-disk meta-data should be exposed through this same API, to improve portability between file systems and OSes. For example, the extended attribute API could be used to expose additional inode bits that are currently manipulated with various `ioctl`s, which each program or OS component must understand.

The VFS can be thought of as an intertwined combination of an engine that dispatches calls to other "lower" file systems and a "library" of generic file system functionality. Functionality that exists directly in VFS methods makes it more difficult to develop layered file systems for two primary reasons. First, more of the VFS must be replicated by the bottom half of layered file systems (as shown in Figure 2). Second, the functionality embedded within the VFS cannot be changed easily, so the layered file system developer must work around it. Therefore:

(ii) A stronger separation should be made between the dispatch engine and the generic file system functionality of the VFS.

For example, currently each system that uses the vnode interface has a single-component `lookup` routine that is driven by a larger VFS lookup routine that handles multi-component lookup operations (the so-called *namei* OS routine). The vnode should have a multi-component lookup operation, that in most cases is serviced by a generic VFS function. For most file systems, this generic function would repeatedly call the single component lookup routine, much as the current VFS does. However, some file systems (e.g., NFSv4 [Shepler et al. 2000]) could take care of the overall multi-component lookup routine to provide better performance (e.g., for constructing a NFSv4 *compound* operation). The key advantage of this architecture is that it adheres to the following two design principles:

(iii) Generic functionality should not be embedded within the VFS.

and

(iv) File systems should be able to control their behavior at all levels.

As soon as a file-system–related system call enters the kernel, it should dispatch the call to a vnode operation to perform the operation. This vnode operation could in turn call other vnode operations. For example, the `mkdir` system call could call a `do_mkdir` vnode operation. In the same way as a generic multi-component `lookup` could call a single component `lookup`, a generic `do_mkdir` would call the proper `lookup`, `lock`, `access`, `mkdir`, and `unlock` vnode methods. By overriding just these generic methods, a layered file system could change VFS functionality. Thus:

(v) Exposing more OS functionality to the file system helps to eliminate missing interception points.

APPENDIX

A. PROTECTION OF VFS OBJECTS' FIELDS

On Linux, the VFS and lower-level file systems access object's fields directly. This poses a problem for layered file systems because they must keep their own objects' fields consistent with the corresponding lower-level object. For example, the file size field of the lower inode must be copied to the upper inode on every file size update.

As described in Section 3, Solaris, FreeBSD, and Windows do not have this problem, because there is no duplication of attributes between the upper and lower-level file systems. The lower and upper layers are separated and can only read or modify each other's properties and data via function calls. On the face of it, wrapping many fields in methods might add noticeable overheads. To measure the actual overheads that would be added if an object-oriented paradigm were included in the Linux kernel we performed the following experiment.

We modified a Linux kernel in such a way that the inode private fields are only accessed directly if no methods for accessing these fields are provided by a file system. If, however, the file system provides methods for reading or updating such fields, then these methods are called instead. A Perl script performs this Linux kernel instrumentation automatically. In particular, the script performs three actions:

(1) The script inserts new methods into the inode operations vector that access the object's fields. For example, in the case of *i_mode*, the script adds the following function declarations to the inode operations structure:

```
umode_t (*read_i_mode)(struct inode *inode);
void (*write_i_mode)(struct inode *inode, umode_t new_val);
```

(2) For each inode field, the script adds inline functions to access the inode's object. If the inode object has a method defined for that field, then the object's own method is

used. Otherwise, the object's field is read directly. For example, the script adds the following function to read the *i_mode* field:

```
static inline umode_t read_i_mode(struct inode *inode)
{
      if (inode->i_op && inode->i_op->read_i_mode)
              return inode->i_op->read_i_mode(inode);
      else
              return inode->i_mode;
}
```

The script defines similar methods for updating other inode fields.

(3) Every access to private inode fields is converted to use one of the inline functions. For example,

```
inode->i_mode = mode;
```

is converted to

```
write_i_mode(inode, (mode));
```

Also, the script properly handles more complicated cases, such as

```
inode->i_nlink++;
```

which are converted into

```
write_i_nlink(inode, read_i_nlink(inode) + 1);
```

Our Perl script is only 133 lines long. We used it to instrument the 2.6.11 Linux kernel. In particular, we protected the following twelve inode private fields: *i_ino*, *i_mode*, *i_nlink*, *i_uid*, *i_gid*, *i_rdev*, *i_size*, *i_blkbits*, *i_blksize*, *i_version*, *i_blocks*, and *i_bytes*. This required converting 3,113 read and 1,389 write operations into function calls. The script added 207 new lines of code and modified 3,838 lines of Linux source code. The compiled Linux kernel size overhead was 22,042 bytes.

To evaluate the CPU time overheads, we ran two benchmarks on a 1.7GHz Pentium IV with 1GB of RAM. The system disk was a 30GB 7,200 RPM Western Digital Caviar IDE formatted with Ext3. The tests were located on a dedicated Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disk formatted with Ext2. We remounted the file systems between each benchmark run to purge file system caches. We ran each test at least ten times and used the Student-*t* distribution to compute the 95% confidence intervals for the mean elapsed, system, and user times. In each case the confidence intervals were less than 5% of the mean.

First, we ran Postmark 1.5 [Katcher 1997], which simulates the operation of electronic mail servers. Postmark performs a series of file system operations such as appends, file reads, creations, and deletions. This benchmark uses little CPU but is I/O intensive. We configured Postmark to create 20,000 files, between 512–10K bytes in size, and perform 200,000 transactions. The rest of the parameters were the Postmark defaults. We ran Postmark under instrumented and vanilla versions of the 2.6.11 Linux kernel. The elapsed times were statistically indistinguishable, and the 95% confidence interval for the system time overhead was between 0.1% and 2.5% (the average was 1.3%).

Second, we ran a custom benchmark that invoked a `stat` system call on one file 100 million times. A `stat` call looks up the file and copies thirteen inode fields into a user buffer. Our script instrumented nine out of these thirteen fields. There is very little I/O during this benchmark, because the same file is used repeatedly (the CPU utilization was 100%). This is a worst case for this system, as the overheads are entirely CPU-bound. Our experiments showed that the instrumented kernel had a 2.3% system time overhead while running this intense sequence of `stat` system calls.

As a result of our experiments, we conclude that the addition of the an object-oriented per-field-access infrastructure into the Linux kernel has negligible impact on the operation of the file systems.

ACKNOWLEDGMENTS

This work has evolved over more than a decade. We would like to thank the thousands of world-wide FiST and Unionfs users—from both academia and industry—who downloaded and used the software, and the many who reported bugs and submitted patches. Special thanks go to Ion Bădulescu, who developed much of the initial Linux stackable templates; to Yevgen Borodin, Michael Gorbovitski, and Haik Lorenz who ported FiST to Linux 2.6 in 2004; and to Sunil Satnur, who stress-tested the code using a POSIX test suite, and then fixed many bugs.

This work was partially made possible by an NSF CAREER award EIA-0133589, NSF Trusted Computing Award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1, all awarded to Stony Brook University. Earlier parts of this work were accomplished at Columbia University, partly under NSF CISE Research Infrastructure grants CDA-90-24735 and EIA-9625374.

REFERENCES

ANDERSON, D., CHASE, J., AND VADHAT, A. 2000. Interposed Request Routing for Scalable Network Storage. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*. USENIX Association, San Diego, CA, 259–272.

APPAVOO, J., AUSLANDER, M., DASILVA, D., EDELSOHN, D., KRIEGER, O., OSTROWSKI, M., ROSENBURG, B., WISNIEWSKI, R., AND XENIDIS, J. 2002. K42 overview. `www.research.ibm.com/K42/`.

ARANYA, A., WRIGHT, C. P., AND ZADOK, E. 2004. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*. USENIX Association, San Francisco, CA, 129–143.

ARPACI-DUSSEAU, A. C. AND ARPACI-DUSSEAU, R. H. 2001. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, Banff, Canada, 43–56.

AYERS, L. 1997. E2compr: Transparent file compression for Linux. *Linux Gazette 18*. `www.linuxgazette.com/issue18/e2compr.html`.

BALZER, R. AND GOLDMAN, N. 1999. Mediating connectors. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*. IEEE, Austin, TX, 72–77.

BLAZE, M. 1993. A Cryptographic File System for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*. ACM, Fairfax, VA, 9–16.

BURNETT, N. C., BENT, J., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2002. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, Monterey, CA, 29–44.

CLUSTER FILE SYSTEMS, INC. 2002. Lustre: A scalable, high-performance file system. `www.lustre.org/docs/whitepaper.pdf`.

CORNER, M. AND NOBLE, B. D. 2002. Zero-Interaction Authentication. In *The Eigth ACM Conference on Mobile Computing and Networking*. ACM, Atlanta, GA, 1–11.

DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2001. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, Banff, Canada, 202–215.

ENGLER, D., KAASHOEK, M. F., AND O'TOOLE JR., J. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*. ACM SIGOPS, Copper Mountain Resort, CO, 251–266.

GHORMLEY, D. P., PETROU, D., RODRIGUES, S. H., AND ANDERSON, T. E. 1998. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the Annual USENIX Technical Conference*. ACM, Berkeley, CA, 39–52.

GOH, E.-J., SHACHAM, H., MODADUGU, N., AND BONEH, D. 2003. Sirius: Securing remote untrusted storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*. Internet Society (ISOC), San Diego, CA, 131–145.

GUY, R. G., HEIDEMANN, J. S., MAK, W., PAGE JR., T. W., POPEK, G. J., AND ROTHMEIER, D. 1990. Implementation of the Ficus replicated file system. In *Proceedings of the Summer USENIX Technical Conference*. IEEE, Anaheim, CA, 63–71.

HALCROW, M. A. 2004. Demands, Solutions, and Improvements for Linux Filesystem Security. In *Proceedings of the 2004 Linux Symposium*. Linux Symposium, Ottawa, Canada, 269–286.

HARDER, B. 2001. Microsoft Windows XP System Restore. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwxp/html/windowsxpsystemrestore.asp`.

HEIDEMANN, J. S. AND POPEK, G. J. 1991. A layered approach to file system development. Tech. Rep. CSD-910007, UCLA.

HEIDEMANN, J. S. AND POPEK, G. J. 1994. File system development with stackable layers. *ACM Transactions on Computer Systems 12,* 1 (February), 58–89.

HEIDEMANN, J. S. AND POPEK, G. J. 1995. Performance of cache coherence in stackable filing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS, Copper Mountain Resort, CO, 3–6.

HOFMEYR, S. A., SOMAYAJI, A., AND FORREST, S. 1998. Intrusion detection using sequences of system calls. *Journal of Computer Security 6*, 151–180.

INDRA NETWORKS. 2004. StorCompress. `www.indranetworks.com/StorCompress.pdf`.

JONES, M. B. 1993. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP '93)*. ACM, Asheville, NC, 80–93.

JOUKOV, N., RAI, A., AND ZADOK, E. 2005. Increasing distributed storage survivability with a stackable raid-like file system. In *Proceedings of the 2005 IEEE/ACM Workshop on Cluster Security, in conjunction with the Fifth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*. IEEE, Cardiff, UK, 82–89. (**Won best paper award**).

KASHYAP, A., DAVE, J., ZUBAIR, M., WRIGHT, C. P., AND ZADOK, E. 2004. Using the Berkeley Database in the Linux Kernel. `www.fsl.cs.sunysb.edu/project-kbdb.html`.

KASHYAP, A., PATIL, S., SIVATHANU, G., AND ZADOK, E. 2004. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*. USENIX Association, Atlanta, GA, 69–79.

KATCHER, J. 1997. PostMark: A New Filesystem Benchmark. Tech. Rep. TR3022, Network Appliance. `www.netapp.com/tech_library/3022.html`.

KEROMYTIS, A. D., PAREKH, J., GROSS, P. N., KAISER, G., MISRA, V., NIEH, J., RUBENSTEIN, D., AND STOLFO, S. 2003. A holistic approach to service survivability. In *Proceedings of the 2003 ACM Workshop on Survivable and Self-Regenerative Systems*. ACM, Fairfax, VA, 11–22.

KHALIDI, Y. A. AND NELSON, M. N. 1993. Extensible file systems in Spring. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP '93)*. ACM, Asheville, NC, 1–14.

KLEIMAN, S. R. 1986. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference*. USENIX Association, Atlanta, GA, 238–247.

KLOTZBÜCHER, M. 2004. Development of a Linux Overlay Filesystem for Software Updates in Embedded Systems. M.S. thesis, Universität Konstanz, Konstanz, Germany.

MACDONALD, J., REISER, H., AND ZAROCHENTCEV, A. 2002. Reiser4 transaction design document. www.namesys.com/txn-doc.html.

MAZIÉRES, D. 2001. A Toolkit for User-Level File Systems. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, Boston, MA, 261–274.

MICROSOFT CORPORATION. 2004a. File System Filter Manager: Filter Driver Development Guide. www.microsoft.com/whdc/driver/filterdrv/default.mspx.

MICROSOFT CORPORATION. 2004b. Installable File System Development Kit. www.microsoft.com/whdc/devtools/ifskit/default.mspx.

MICROSOFT CORPORATION. 2004c. Microsoft MSDN WinFS Documentation. http://msdn.microsoft.com/data/winfs/.

MIRETSKIY, Y., DAS, A., WRIGHT, C. P., AND ZADOK, E. 2004. Avfs: An On-Access Anti-Virus File System. In *Proceedings of the 13th USENIX Security Symposium (Security 2004)*. USENIX Association, San Diego, CA, 73–88.

MUNISWAMY-REDDY, K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. 2004. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*. USENIX Association, San Francisco, CA, 115–128.

NAGAR, R. 1997. *Windows NT File System Internals: A developer's Guide*. O'Reilly, Sebastopol, CA, 615–667. Section: Filter Drivers.

NETWORK ASSOCIATES TECHNOLOGY, INC. 2004. McAfee. www.mcafee.com.

NUGENT, J., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. 2003. Controlling Your PLACE in the File System with Gray-box Techniques. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, San Antonio, TX, 311–323.

ONEY, W. 2003. *Programming the Microsoft Windows Driver Model*, Second ed. Microsoft Press, Redmond, WA.

PENDRY, J. S. AND MCKUSICK, M. K. 1995. Union mounts in 4.4BSD-Lite. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*. USENIX Association, New Orleans, LA, 25–33.

PHILLIPS, D. 2001. A directory index for EXT2. In *Proceedings of the 5th Annual Linux Showcase & Conference*. USENIX Association, Oakland, CA, 173–182.

ROSENTHAL, D. S. H. 1990. Evolving the Vnode interface. In *Proceedings of the Summer USENIX Technical Conference*. USENIX Association, Anaheim, CA, 107–118.

ROSENTHAL, D. S. H. 1992. Requirements for a "Stacking" Vnode/VFS interface. Tech. Rep. SD-01-02-N014, UNIX International.

SCHAEFER, M. 2000. The migration filesystem. www.cril.ch/schaefer/anciens_projets/mfs.html.

SELTZER, M. AND YIGIT, O. 1991. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*. USENIX Association, Dallas, TX, 173–184.

SHANAHAN, D. P. 2000. CryptosFS: Fast cryptographic secure nfs. M.S. thesis, University of Dublin, Dublin, Ireland.

SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., BEAME, C., EISLER, M., AND NOVECK, D. 2000. NFS Version 4 Protocol. Tech. Rep. RFC 3010, Network Working Group. December.

SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2003. Semantically-Smart Disk Systems. In *Proceed-*

ings of the Second USENIX Conference on File and Storage Technologies (FAST '03). USENIX Association, San Francisco, CA, 73–88.

SKINNER, G. C. AND WONG, T. K. 1993. "Stacking" Vnodes: A progress report. In *Proceedings of the Summer USENIX Technical Conference*. USENIX Association, Cincinnati, OH, 161–174.

SMCC. 1992. *lofs – loopback virtual file system*. Sun Microsystems, Inc. SunOS 5.5.1 Reference Manual, Section 7.

SOLOMON, D. A. AND RUSSINOVICH, M. E. 2000. *Inside Microsoft Windows 2000*. Microsoft Press, Redmond, WA.

SOPHOS. 2004. Sophos Plc. `www.sophos.com`.

SUNSOFT. 1994. Cache file system (CacheFS). Tech. rep., Sun Microsystems, Inc. February.

SYMANTEC. 2004. Norton Antivirus. `www.symantec.com`.

SZEREDI, M. 2005. Filesystem in Userspace. `http://fuse.sourceforge.net`.

TZACHAR, N. 2003. SRFS kernel module. Tech. rep., Computer Science Department, Ben Gurion University. September. `www.cs.bgu.ac.il/~srfs/publications/implementation_report.pdf`.

WRIGHT, C. P., DAVE, J., GUPTA, P., KRISHNAN, H., QUIGLEY, D. P., ZADOK, E., AND ZUBAIR, M. N. 2006. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS) 2,* 1 (February), 1–32.

WRIGHT, C. P., MARTINO, M., AND ZADOK, E. 2003. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, San Antonio, TX, 197–210.

ZADOK, E. 2001. FiST: A System for Stackable File System Code Generation. Ph.D. thesis, Computer Science Department, Columbia University. `www.fsl.cs.sunysb.edu/docs/zadok-phd-thesis/thesis.pdf`.

ZADOK, E., ANDERSON, J. M., BĂDULESCU, I., AND NIEH, J. 2001. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, Boston, MA, 289–304.

ZADOK, E. AND BĂDULESCU, I. 1999. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*. Raleigh, NC, 141–151.

ZADOK, E., BĂDULESCU, I., AND SHENDER, A. 1999. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, Monterey, CA, 57–70.

ZADOK, E. AND NIEH, J. 2000. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*. USENIX Association, San Diego, CA, 55–70.

ZADOK, E., OSBORN, J., SHATER, A., WRIGHT, C. P., MUNISWAMY-REDDY, K., AND NIEH, J. 2004. Reducing Storage Management Costs via Informed User-Based Policies. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*. IEEE, College Park, MD, 193–197.

Received October 2005; accepted December 2005