User Space Storage System Stack Modules with File Level Control

Sumit Narayan
Dept. of E.C.E.
Univ. of Connecticut, USA
sumitn@engr.uconn.edu

Rohit K. Mehta
E.C.S., School of Engineering
Univ. of Connecticut, USA
rohitm@engr.uconn.edu

John A. Chandy
Dept. of E.C.E.
Univ. of Connecticut, USA
chandy@engr.uconn.edu

Abstract

Filesystem in Userspace (FUSE) is a typical solution to simplifying writing a new file system. It exports all file system calls to the user-space, giving programmer the ability to implement actual file system code in the user-space but with a small overhead due to context switching and memory copies between the kernel and the user-space. FUSE, however, only allows writing non-stackable file systems. The other alternative to simplify writing file system code is to use File System Translator (FiST), a tool that can be used to develop stackable file systems using template code. FiST is limited to the kernel space and requires learning a slightly simplified file system language that describes the operation of the stackable file system. In this work, we combine FUSE with FiST and present a stackable FUSE module which will allow users to write stackable file systems in the user-space. To limit the overhead of context switching operations, we provide this module in combination with our previously developed ATTEST framework that provides ways to filter files so that only those with specific extended attributes are exported to the user-space daemon. Further, these attributes can also be exported to user-space where multiple functions can behave as stackable modules with dynamic ordering. Another advantage of such a design is that it allows non-admin users to have stackable file system implemented and mounted, for example, on their respective home directories. In our experiments, we observe that having stackable modules in user-space has an overhead of around 26% for writes and around 39% for reads when compared to the standard stackable file systems.

1 Introduction

The file system is often seen as one of the most critical part of an operating system. It handles the task of storing and organizing user files and their data on the underlying storage devices. It is comprised of very complex C kernel code which takes several months to develop and stabilize and is usually written for a particular operating system platform. The file system code must interact with the operating system's virtual file system manager to receive system calls from the user-space, with virtual memory manager for page allocation and memory management within the kernel and with the virtual device layer to communicate with the storage devices and store data. This makes the file system code remarkably complex to understand and very hard to develop. An average modern file system is comprised of around 50,000–60,000 lines of code and supports a variety of features, such as B-tree based search, flexible data extents, access control lists, extended attributes, etc. [28]. This low-level kernel code is very difficult to program and is often the origin of bugs in a storage system [9, 17]. To add any new feature in a file system, a programmer needs to have a thorough understanding and working of the file system. Apart from programming, providing support and maintenance for such large and complex file systems with several features and diverse mount options is also very hard. Thus, file system development and maintenance is always considered to be the work of the select few who have a very deep knowledge of the file system and also the operating system.

Several techniques have been suggested to simplify the process of file system development. To address the need to quickly develop and incorporate new features in an existing file system, the Linux kernel has provisions for implementing stackable file systems. Stackable file systems [21, 10] give developers a quicker way to add new features to a file system through an extensible file system interface. It reduces the complexity of developing a newer file system, in that it allows features to be added incrementally in steps instead of creating a new file system from scratch, or modifying an existing one. However, to obtain the best performance, these file systems

are tightly integrated into the Linux kernel or are designed and developed to run as a kernel module, thus requiring the uphill task of understanding the kernel before starting to develop a file system. File System Translator (FiST) [27] is a file system generation tool that simplifies the task of creating stackable file systems by generating most of the code from a standard file system template. The programmer is required to provide code only for the main functionality of the file system, which is then called from another code written in FiST language and fed to the FiST file system generation tool. The resulting code can then be inserted into a live system as a loadable kernel module. However, the simplified coding now requires learning a new file system template language.

Kernel-space file systems are not always the best way to develop a file system and suffer from several drawbacks. They cannot be ported across different platforms and they also do not provide any options for non-privileged users to mount a file system. File system in User Space (FUSE) is another solution to simplify writing a file system and can be ported across different operating system platforms. It has been integrated into the Linux kernel tree and has ports available for other major operating systems. FUSE exports all file system calls within the kernel to the user-space through a simple application programming interface (API) by connecting to a daemon that is running in the user-space. FUSE provides a good way to write virtual file systems, in that the file systems do not store any data themselves. Writing a file system in user-space is several folds easier when compared to writing a kernel-space file system. FUSE also has provisions to permit non-privileged users to mount FUSE-based file systems. These user-space file systems however come with a small overhead due to context switches and memory copies made during the data transfer operations [25, 26].

As discussed above, FiST and FUSE are two very common solutions to simplify the process of writing file systems. But, as observed, performance, portability and availability to non-privileged users, all cannot be achieved together. In this work we propose a stackable FUSE architecture that will allow developing stackable file systems in user-space. And to limit the overhead due to context switching between the kernel space and user-space, we propose to combine this stackable FUSE design with ATTEST, an attribute-based storage framework that allows defining policies to filter files and thus

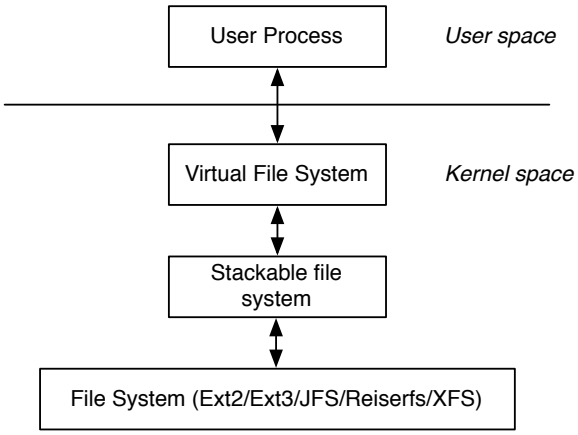

Figure 1: Stackable file systems.

file system operations on a per-file or per-directory basis [15].

The remainder of this paper is organized as follows. Section 2 discusses the stackable file system model, while Section 3 describes the FUSE architecture. Section 4 describes the previously developed ATTEST framework for an extendable storage system. In Section 5, we provide details of our design for stackable FUSE module, Section 6 gives a brief description of our implementation approach and Section 7 shows the performance results. Section 8 is an overview of the related work. We end this paper with a section on our conclusions and proposed future work.

2 Stackable File Systems

The idea of stackable or layered file systems was adapted from the *vnode* interface first implemented on SunOS in 1984 [12]. Stackable file systems [20, 21, 24, 22] are stand-alone file systems that can be mounted on top of an existing file system mount point. Figure 1 shows the typical arrangement of a stackable file system present between the Virtual File System (VFS) and a lower-level file system, which may or may not be a device-based file system. The advantage of developing a stackable file systems is that they can be used to extend the functionality of an existing file system without changing the code of the original file system. A stackable file system creates a *vnode* with its own operations that is inserted on top of the *vnode* belonging to the underlying file system. This allows a stackable file system to perform operations in between the VFS and the lower file system calls. For example, an encryption process

can take place before the data is written on the lower file system, or, a decryption function can be run after the data is read from the lower file system. Stackable file systems can be used to add many functionalities such as compression, encryption, caching, etc. to an existing file system. Other examples of stackable file systems include WrapFS [26], UnionFS [18], RAIF [11], AVFS [14], etc.

fistgen is a set of File System Translator (FiST) language and tool that allows a developer to create a stackable file system by only describing the core functionalities of the file system [27]. The file system generator tool, generates the code for a file system that can be directly loaded as a kernel module into a live Linux system. To add some of the functionalities in FiST, however, requires learning a new language. Since loading a kernel module in a system is restricted to privileged users only, file systems generated using FiST can only be used if inserted into the system previously by an administrator or a privileged user.

3 File system in User space (FUSE)

Filesystem in Userspace (FUSE) is a combination of a user-space library and a kernel module for Unix-like operating systems that allows non-privileged users to create their own file systems without editing the kernel code [2]. This is achieved by running the file system code in user-space, while the FUSE module only provides a bridge to the actual kernel interfaces through a set of APIs. FUSE's kernel module simply redirects the Virtual File System (VFS) calls to the user-space daemon. Figure 2 shows the internal architecture of FUSE. Several FUSE-based file systems are already in common use. FUSE can be particularly useful in providing a POSIX interface for files which are accessible over the network through different network protocols. Some of the file systems based on such a design are *sshfs* [6], *httpfs* [3], *CurlFtpFs* [1], etc. FUSE file systems are easier to maintain since they run in user-space. They are also easier to code and debug compared to the kernel file systems. Running file system in user-space also implies access to more libraries. Thus, FUSE file systems can be written in any language that has a binding to the FUSE libraries, including Ruby and Python.

However, file systems created using FUSE are always the lowest file system in the storage stack. This means that all FUSE requests must return from the FUSE

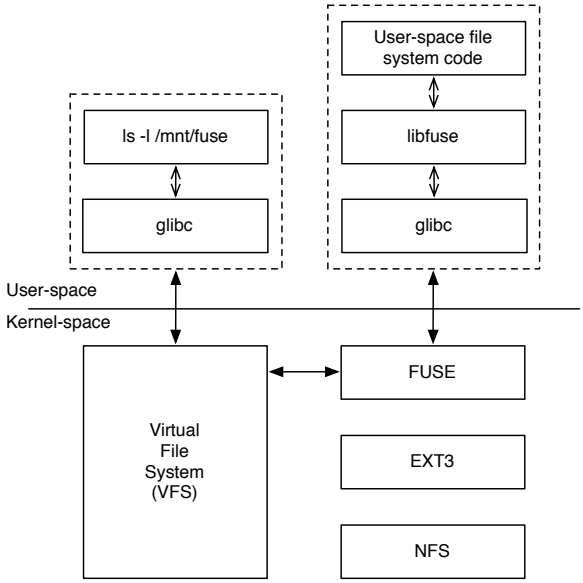

Figure 2: FUSE architecture showing kernel module and user-space library.

layer to the user-space applications without going to the lower-level file system within the kernel, similar to a stackable file system. Thus, FUSE, in its current form is not a solution for developing a stackable user-space file system.

4 ATTribute-based Extendable STorage (ATTEST)

ATTEST is an attribute-based extendable storage framework that allows policy decisions to be made at file-level granularity and at all levels of the storage stack through the use of file's, or directory's extended attributes [15]. These attributes can be used to enable or disable stackable file systems thus behaving like plugins and also allow the user to define rules to identify redundancy or throughput requirements on per-file basis to select the device for storing data. ATTEST allows user to set define file-based rules or directory-based policies that will selectively enable or disable options at each layer within the storage system stack. The rules set on a file are stored in the file's extended attributes and move with the file while directory-based policies are set on all files created under it. By allowing per-file attribute-based policies, it becomes possible to implement storage policies on a much smaller granularity. The ATTEST framework also pushes these attributes to the operating systems' storage device layer called the *logical volume manager*

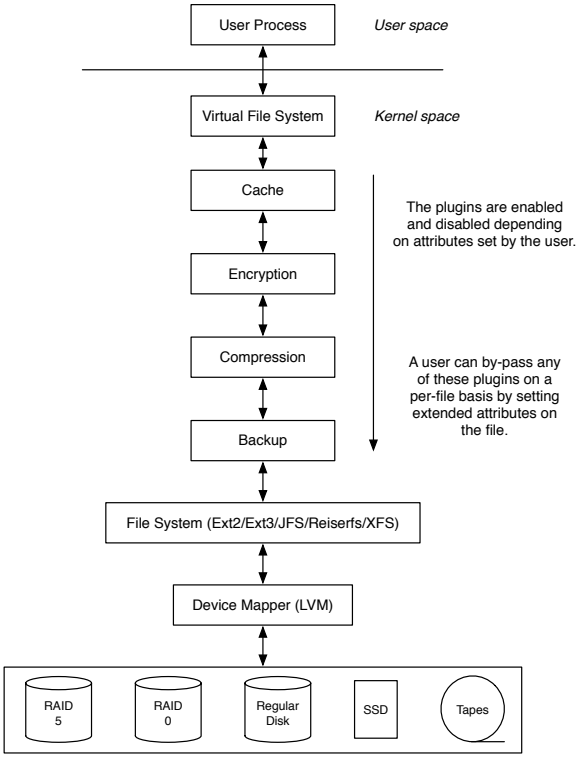

Figure 3: ATTEST architecture with cache, encryption, compression, backup plugins and RAID-5, RAID-0, regular disk, SSD and tapes.

(LVM). This allows the user to also pass a file's properties and its data redundancy or throughput requirements. ATTEST framework will then select the device based on those attributes to store data blocks belonging to that file.

The ATTEST framework was designed with the objective to include computationally expensive, but necessary functionalities at the file system layer, or in the device manager layer under a single file system mount-point but with user-controlled rules or policies to determine which files or directories will really be applying them. Such a scheme would allow including functionalities in file systems which are otherwise typically ignored simply to avoid distributing their computation overhead across all files present on the file system. Typical examples of such functionalities include encryption, compression, redundancy, etc.

5 Stackable FUSE

As already mentioned, FiST and FUSE are techniques to design new file systems with lower learning curve as

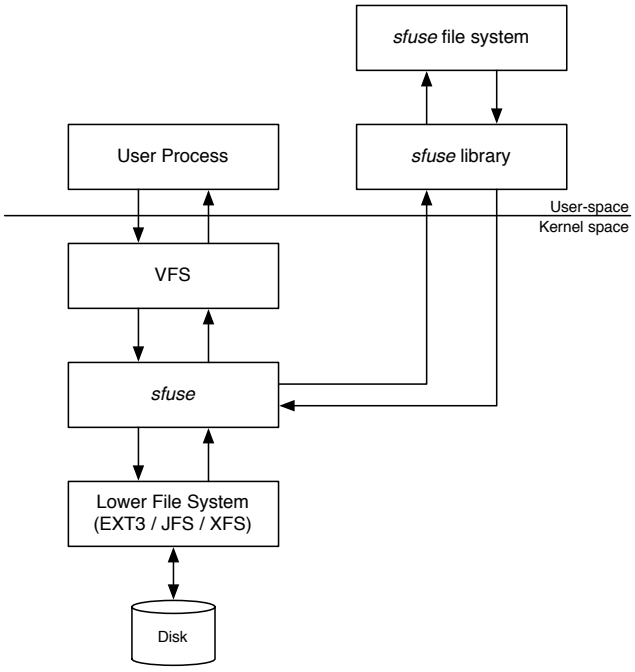

Figure 4: Operation flow in *sfuse* with a user-space *sfuse* file system.

compared to writing a standard kernel-level file system from scratch. FiST helps by extending functionalities of an existing file system while FUSE allows easy programming and maintenance of file system in the user-space. FUSE also provides the ability for non-privileged users to mount file systems and use it.

In this work, we propose a new stackable file system module called *sfuse* that will provide users with a FUSE-like interface in user-space to write their own file system in the user-space. The added advantage of *sfuse* and the difference compared to FUSE is that it will provide stackability similar to that available using FiST. Thus, data in all I/O operations will be sent to the user-space, copied or modified, and returned to kernel-space to be pushed to the lower-level file system. Since a FUSE-based file system have a cost due to context switching and memory copies, we plan to extend the idea of ATTEST and limit the overhead only to files that require the user-space functionality implemented in the file system.

There are several advantages of porting file system stackability to the user-space. One of the main advantages is that such a scheme would allow any user on the machine to mount a stackable file system without the need of administrator privileges. Along with stackability, per-file control on the files will allow the user more

control on how the files are treated. *sfuse* also avoids the need for the user to understand the FiST language in developing a stackable file system. Figure 4 shows the operation flow with a *sfuse*-based file system present in the user-space. In case of READ, the data would flow away from the lower file system, while during WRITE operation, the data would flow towards the lower-level file system. Section 6 provides more detail on the internals of the *sfuse* file system module.

6 Implementation

We implemented our stackable FUSE-like file system *sfuse* on Linux kernel version 2.6.24. We started by first creating a stackable base file system *basefs* using FiST. We used a patched source of FiST version 0.2.1 to create the stackable file systems which was compatible with the Linux kernel version present on our machine. *sfuse* is designed to export all file system operations to the user-space daemon, similar to the default FUSE module. To export I/O functions to the user-space daemon, we modified the user-space FUSE library to receive requests from the kernel even without any previous file OPEN operation. All I/O requests are forwarded to the user-space file system, irrespective of whether an OPEN operation was performed on that file. This is in contrast to FUSE, where an I/O operation can be performed only after an OPEN call is made. This step is required in FUSE to open the actual file on the ported file system and obtain a file handle in the user-space. The file handle information is later used in identifying the file on which I/O needs to be performed. In *sfuse*, the file is actually opened within the *sfuse*'s Linux kernel module and multiple operations on the same file are handled within the kernel module.

A user may however also opt to also use the exported OPEN function in the user space, depending on the requirement of the stacked file system. As an example, for a simple encryption file system, encode and decode functions can be run without having any knowledge of the file handle. In another case, if the stackable file system is designed to count number of times a file is opened, OPEN functions will have to be implemented in the user-space.

To implement stackability in FUSE, we also require the user-space FUSE library to return the request data buffer back to the kernel after performing the stack function. This is done in the same way as any write operation would be performed in FUSE module. File systems developed based on *sfuse* are mounted in the same way as a FUSE-based file system. The mount binary file requires two parameters – the mount point directory and the directory which needs to have the stacked functionality on top of it. In the kernel, data structures for storing the file system's private information must include information regarding the lower-level file system along with the connection pointers of the user-space FUSE daemon.

The user can control the files which must be exported to the user-space by using ATTEST. The user can lay rules or set policies for each file by directly setting the file's extended attributes, or by including the rules in the ATTEST config file. More details on how to set the rules and policy in an ATTEST framework is explained in [15]. Stackable FUSE also allows attributes set by the ATTEST framework to be passed from the kernel-space to the user-space as tags along with any I/O request. This will allow the user to perform dynamic ordering of multiple stacked functions in the user-space without going back into the kernel-space.

Our current implementation only supports synchronous operations. This means that all operations can return to the kernel space only after the user-space functions have returned. As part of our future work, we plan to support asynchronous operations in *sfuse* library, which will allow the requests to be appended to a queue in the user-space file system. This queue will be cleaned by a thread running continuously on the system. One place where such a mechanism can be very useful is in performing lazy data backup and deduplication on a per-file or a per-directory basis. With the assistance of ATTEST rules, the user can also define policies, such as, if the files need to backed up after compression, or encryption, or neither.

7 Results

We evaluated the performance of *sfuse* by running IOZone [4], a popular benchmarking tool that performs synthetic read/write tests to determine the throughput of the system over a variety of file system configurations. We conducted our experiments on a 1.8 GHz dual-core dual processor AMD Opteron machine with 2 GB RAM and two 40 GB hard disk drives running Linux kernel 2.6.24. The experiment was run for a file size of 2 GB with record size set to 128 KB. Figure 5 shows the results with overhead of using *sfuse* compared to other file system configurations.

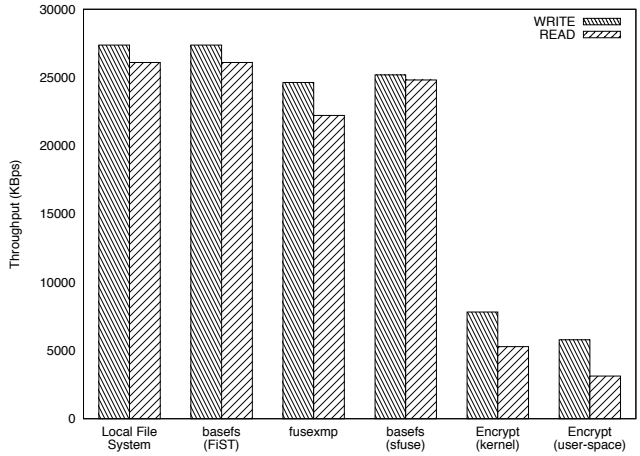

Figure 5: Throughput comparison for different setup of file system using IOZone benchmarking tool on a 2 GB file and record size of 128 KB.

We first ran IOZone on a default EXT3 formatted non-root partition to obtain the base performance of the system without any stacks. Next, we ran IOZone on *basefs*, a stackable file system generated by FiST. *basefs* is an empty file system in that it simply forwards all calls to the lower-level file system. By running IOZone on *basefs*, we evaluate the overhead of including a stackable file system between VFS and disk-based file system such as EXT3. *basefs* is available along with the source code of *fistgen*. From Figure 5, we can see that including an empty stackable file system has a very negligible overhead.

Our third experiment was run on a *fusexmp* file system mount. *fusexmp* is a FUSE-based file system that simply mirrors the root directory of the system on the mounted directory. It is available freely along with the FUSE source code [2]. Our experiments confirm the overhead that is expected to be present on any FUSE-based file system due to memory copies between user-space and the kernel-space. We observed almost 10% overhead for writes and slightly less than 15% for reads in this test. Our fourth setup was an empty file system similar to *basefs*, but set up in *sfuse*. This file system simply returned the request back to the kernel space without doing anything to the transferred data. We conducted this experiment to observe the real overhead of context switching and memory copy operations. In our experiments, we observed an overhead of around 8% for writes and around 5% for reads when compared to the local file system.

Our fifth experiment was done on a slightly modified version of *cryptfs* stackable file system available with the *fistgen* source code. We disabled the encryption of file names in *cryptfs* and only allowed data block encryption. We implemented the same encryption algorithm in user-space and implemented a *sfuse* file system for it. We observed almost 26% overhead for writes and around 39% overhead for reads by porting the code to the user-space. The overhead in this case is primarily because of context switching between the user-space and kernel space. The data buffer memory is copied two times in each I/O operation for each direction of the data flow, i.e., once from the kernel to the user-space, and then, from the user-space back to the kernel space. This overhead also comes from several other aspects within the operating system like processor registers that need to be saved and restored, cache entries that need to be evicted and reloaded for the incoming processes, etc. [13, 19].

While the overhead in our experiments are certainly non-negligible and casts doubt over the need to port file systems into the user-space, we remind the reader the benefits such as the ability for non-administrator accounts to control their data, simpler programming and debugging in user-space with FUSE bindings available in many programming languages other than C and per-file granularity control sufficient to make this a useful solution. Further, our *sfuse* code has not been highly optimized and could be improved significantly to lower this overhead.

8 Related Work

Significant effort has been put into providing users with control of their data in terms of where their data is placed in the storage system. Redundant Array of Independent Filesystems (RAIF) [11], for example, is a stackable file system that allows user to define rules to determine data placement policy. RAIF allows users to distribute data across different file systems and define redundancy across it. UmbrellaFS [8] is another solution that allows the users to define distribution policy, but across different devices with each device having their own redundancy and throughput limitation. ATTEST differs from these existing solutions, in that ATTEST allows rules that enable or disable stackable plugins mounted in the system by the administrator and also allows the user to define rules on data placement by selecting underlying storage devices, each with their own redundancy or throughput characteristics.

Apart from FUSE, there also exist other solutions to writing file systems in user-space. UserFS [7] was an idea proposed in 1993 which exported file system requests to the user-space through a file descriptor. *puffs* is an export of FUSE-like library on the NetBSD operating system [5].

However, all these solutions limit themselves by either allowing the user only to decide where the data is stored or not providing stackability in the file system. To the best of our knowledge, we know of no solution that provides the users a mount point interface that will allow the user to modify or copy the data in a stackable fashion without the need for having any root-privileges.

9 Conclusions and Future Work

In this paper, we have presented a stackable user-based file system model which can be controlled based on user defined rules. Most of the existing file systems make compromise on adding costly functionalities because there is no way to make policy decisions at a finer granularity. In this work, we reinforce our commitment to providing users more control over policy decisions on the files by using ATTEST. Stackable file systems in user-space opens up a variety of opportunities to design file systems. However, the stackability of these file systems are expensive due to context switching. By using the ATTEST framework, we can select files and enable stackability and absorb the overheads only for files that require the stackable functions enabled.

In the future, we plan to implement a stackable module in the LVM layer that will send data blocks present at the disk level to the user-space daemon in a similar fashion. This can be used to write user-spaced disk block manipulation functions such as data deduplication or snapshotting. Such a design will also allow users to write their own disk layout algorithms [23, 16] from the user space.

References

[1] CurlFtpFS - a FTP filesystem based on cURL and FUSE. `http://curlftpfs.sourceforge.net/`.

[2] FUSE: Filesystem in user space. `http://fuse.sourceforge.net`.

[3] HTTP filesystem. `http://httpfs.sourceforge.net/`.

[4] IOZone. `http://www.iozone.org`.

[5] puffs - pass-to-userspace framework file system. `http://www.netbsd.org/docs/puffs/`.

[6] SSH filesystem. `http://fuse.sourceforge.net/sshfs.html`.

[7] Jeremy Fitzhardinge. Userfs. `http://www.goop.org/~jeremy/userfs/`.

[8] John A. Garrison and A. L. Narasimha Reddy. Umbrella file system: Storage management across heterogeneous devices. In *ACM Transactions on Storage*, volume 5, New York, USA, March 2009.

[9] Jim Gray. A census of tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, October 1990.

[10] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. In *ACM Transactions on Computer Systems*, volume 12, pages 58–89, February 1994.

[11] Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok. RAIF: Redundant array of independent filesystems. In *IEEE Conference on Mass Storage Systems and Technologies*, pages 199–214, San Diego, CA, September 2007.

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

[13] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Workshop on Experimental Computer Science*, San Diego, CA, 2007.

[14] Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok. Avfs: An on-access anti-virus file system. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[15] Sumit Narayan and John A. Chandy. ATTEST: ATTributes-based Extendable STorage. *Journal of Systems and Software*, 83(4):548–556, April 2010.

[16] James A. Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Controlling your PLACE in the file system with gray-box techniques. In *Proceedings of the Annual USENIX Technical Conference*, pages 311–323, San Antonio, TX, June 2003.

[17] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 206–220, Brighton, UK, October 2005.

[18] David Quigley, Josef Sipek, Charles P. Wright, and Erez Zadok. UnionFS: User- and community-oriented development of a unification file system. In *Proceedings of 2006 Ottawa Linux Symposium*, pages 349–362, Ottawa, Canada, June 2006.

[19] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 25th ACM Symposium on Applied Computing*, Sierre, Switzerland, March 2010.

[20] David S. H. Rosenthal. Evolving the vnode interface. In *Proceedings of the USENIX Technical Conference*, pages 107–118, Anaheim, CA, Summer 1990.

[21] David S. H. Rosenthal. Requirements for a "stacking" vnode/VFS interface. Technical Report SD-01-02-N014, UNIX International, 1992.

[22] Josef Sipek, Yiannis Pericleous, and Erez Zadok. Kernel support for stackable file systems. In *Proceedings of 2007 Ottawa Linux Symposium*, Ottawa, Canada, June 2007.

[23] Jun Wang and Yiming Hu. PROFS: Performance-oriented data reorganization for log-structured file system on multi-zone disks. In *Proceedings of IEEE/ACM 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 285–293, Cincinnati, OH, August 2001.

[24] Erez Zadok and Ion Badulescu. A stackable file system interface for Linux. In *LinuxExpo 99*, pages 141–151, May 1999.

[25] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.

[26] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, Monterey, CA, June 1999.

[27] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. In *Proceedings of 2000 USENIX Annual Technical Conference*, pages 55–70, San Diego, CA, June 2000.

[28] Erez Zadok, Vasily Tarasov, and Priya Sehgal. The case for specialized file systems, or, fighting file system obesity. *;login: The USENIX Magazine*, 35(1):38–40, February 2010.