Terra Incognita: On the Practicality of User-Space File Systems

Vasily Tarasov[†*], Abhishek Gupta[†], Kumar Sourav[†], Sagar Trehan[†+], and Erez Zadok[†]
Stony Brook University[†], IBM Research—Almaden[*], Nimble Storage[+]

Abstract

To speed up development and increase reliability the Microkernel approach advocated moving many OS services to user space. At that time, the main disadvantage of microkernels turned out to be their poor performance. In the last two decades, however, CPU and RAM technologies have improved significantly and researchers demonstrated that by carefully designing and implementing a microkernel its overhead can be reduced significantly. Storage devices often remain a major bottleneck in systems due to their relatively slow speed. Thus, user-space I/O services, such as file systems and block layer, might see significantly lower relative overhead than any other OS services. In this paper we examine the reality of a partial return of the microkernel architecture—but for I/O subsystems only. We observed over 100 user-space file systems have been developed in recent years. However, performance analysis and careful design of user-space file systems were disproportionately overlooked by the storage community. Through extensive benchmarks we present Linux FUSE performance for several systems and 45 workloads. We establish that in many setups, FUSE already achieves acceptable performance but further research is needed for file systems to comfortably migrate to user space.

1 Introduction

Modern general-purpose OSes, such as Unix/Linux and Windows, lean heavily towards the monolithic kernel architecture. In the 1980s, when the monolithic kernel approach became a development bottleneck, the idea of microkernels arose [1, 15]. Microkernels offer only a limited number of services to user applications: process scheduling, virtual memory management, and Inter-Process Communication (IPC). The rest of the services, including file systems, were provided by user-space daemons. Despite their benefits, microkernels did not succeed at first due to high performance overheads caused by IPC message copying, excessive number of system call invocations, and more.

Computational power has improved significantly since 1994 when the acclaimed Mach 3 microkernel project was officially over. The 1994 Intel Pentium performed 190 MIPS; the latest Intel Core i7 achieves 177,000 MIPS—almost 1,000× improvement. Similarly, cache sizes for these CPUs increased 200× (16KB vs. 3MB). A number of CPU optimizations were added to improve processing speeds, e.g., intelligent prefetchers and a special *syscall* instruction. Finally, the average number of cores per system is up from one to dozens.

At the same time, three factors dominated the storage space. First, HDDs, which still account for most of the storage devices shipped, have improved their performance by only 10× since 1994. Second, dataset sizes increased and data-access patterns became less predictable due to higher complexity of modern I/O stack [19]. Thus, many modern workloads bottleneck on device I/O; consequently, OS services that involve I/O operations might not experience as high performance penalty as before due to CPU-hungry IPC. However, the third factor—the advent of Flash-based storage—suggests the contrary. Modern SSDs are over 10,000× faster than 1994 HDDs; consequently, OS I/O services might have added more relative overhead than before [5]. As such, there is no clear understanding on how the balance between CPU and storage performance shifted and impacted the overheads of user-space I/O services.

In the 90s, Liedtke et al. demonstrated that by carefully designing and implementing a microkernel with performance as a key feature, microkernel overheads could be reduced significantly [7]. Alas, this proof came too late to impact server and desktop OSes, which comfortably settled into the monolithic approach. Today, microkernels such as the L4 family are used widely, but primarily in embedded applications [10]. We believe that the same techniques that allow L4 to achieve high performance in embedded systems can be used to migrate I/O stack in general-purpose OSes to user space.

File systems and the block layer are two main I/O stack OS services. File systems are complex pieces of software with many lines of code and numerous implementations. Linux 3.6 has over 70 file systems, consuming almost 30% of all kernel code (excluding architecture-specific code and drivers). Software Defined Storage (SDS) paradigms suggest moving even more storage-related functionality to the OS. Maintaining such a large code base in a kernel is difficult; moving it to user space would simplify this task significantly, increase kernel reliability, extensibility, and security. In recent years we observed a rapid growth of user-space file systems, e.g., over 100 file systems are implemented using FUSE [18]. Initially, FUSE-based file systems offered a common interface to diverse user data (e.g., archive files, ftp servers). Nowadays, however, even traditional file systems are implemented using FUSE [21]. Many enterprise distributed file system like PLFS and GPFS are implemented in user space [4].

Despite the evident renaissance of user-space file sys-

tems, performance analysis and careful design of FUSE-like frameworks were largely overlooked by the storage community. In this position paper, we evaluate user-space file system's performance. Our results indicate that for many workloads and devices user-space file systems can achieve acceptable performance already now. However, for some setups, the FUSE layer still can be a significant bottleneck. Therefore, we propose to jump-start research on user-space file systems. Arguably, with a right amount of effort, the entire I/O stack, including block layers and drivers, can be effectively moved to user space in the future.

2 Benefits of User-Space File Systems

Historically, most file systems were implemented in the OS kernel, excluding some distributed file systems that needed greater portability. When developing a new file system today, one common assumption is that to achieve production-level performance, the code should be written in the kernel. Moving file system development to user space would require significant change in the community's mindset. The benefits must be compelling: low overheads if any and significantly improved portability, maintainability, and versatility.

Development ease. The comfort of software development (and consequently its speed) is largely determined by the completeness of a programmer's toolbox. Numerous user-space tools for debugging, tracing, profiling, and testing are available to user-level developers. A bug does not crash or corrupt the state of the whole system, but stops only a single program; this also often produces an easily debuggable error message. After the bug is identified and fixed, restarting the file system is as easy as any other user application. Programmers are not limited to a single programming language (usually C or C++), but can use and mix any programming and scripting languages as desired. A large number of useful software libraries are readily available.

Reliability and security. By reducing the amount of code running in the kernel, one reduces the chances that a kernel bug crashes an entire production system. In recent years, malicious attacks on kernels have become more frequent than on user applications. As the main reason for this trend Kermerlis et al. lists the fact that applications are better protected by various OS-protection mechanisms [9]. Counter-intuitively then, moving file system code to the user space makes them more secure.

Portability. Porting user-space file systems to other OSes is much easier than kernel code. This was recognized by some distributed file systems, whose clients often run on multiple OSes [20]. E.g., if file systems were written in user space initially, Unix file systems could have been more readily accessible under Windows.

Performance. The common thinking that user-space file systems cannot be faster than kernel counterparts mistakenly assume that both use the same algorithms and data structures. However, an abundance of libraries is available in the user space, where it is easier to use and try new, more efficient algorithms. For example, predictive prefetching can use AI algorithms to adapt to a specific user's workload; cache management can use classification libraries to implement better eviction policies. Porting such libraries to the kernel would be daunting.

3 Historical Background

The original microkernel-based OSes implemented file systems as user-space services. This approach was logically continued by the modern microkernels. E.g., GNU Hurd supports ufs, ext2, isofs, nfs, and ftpfs file servers [8]. In this case, implementing file systems as user processes was just a part of a general paradigm.

Starting from the 1990s, projects developing user-space file systems as part of monolithic kernels began to appear. These endeavors can be roughly classified into two types: *specialized solutions* and *general frameworks*. The first type includes the designs in which some specific file system was completely or partially implemented in the user space, without providing a general approach for writing user-space file systems. E.g., Steere et al. proposed to move Coda file system's cache management—the most complex part of the system—to the user space [17]. Another example of a specialized solution is Arla—an open source implementation of AFS [20]. Only 10% of Arla's code is located in the kernel which allowed it to be ported to many OSes: Solaris, FreeBSD, Linux, AIX, Windows, and others.

The second type of solutions—general solutions—include the designs that focus explicitly on expanding OS functionality by building a full-fledged frameworks for creating user-space file systems [2, 3, 12]. The original intention behind creating such frameworks was to enable programs to look inside archived files, access the remote files over FTP, and similar use cases—all without modifying the programs. Over time, however, it became clear that there were many more use cases for the user-space file system frameworks. E.g., one can extend a functionality of an existing file system by stacking a user-space file system on top. Lessfs and SDFS add deduplication support to any file system this way [11, 16]. In recent years, some traditional disk-based file systems were ported to user space [13, 21].

It became evident that good support for user-space file systems is needed in the kernel. The Linux FUSE project [18] was a spin-off of AVFS [3] and is currently considered a state of the art in the field. It was merged to Linux mainline starting in version 2.6.14 in 2005. Soon afterwards, FUSE was ported to almost every widely

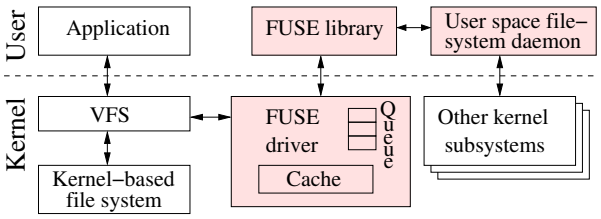

Figure 1: FUSE high-level design.

used general purpose OS: FreeBSD, NetBSD, Mac OS X, Windows, Solaris, and Minix.

Since FUSE was merged in the kernel mainline more than a hundred file systems were developed for it. For comparison, for 23 years of Linux development, about 70 file systems were developed—at a rate of about four times slower than with FUSE. Despite FUSE's rapid adoption, there were few systematic studies on FUSE performance. In fact, to the best of our knowledge, there is only one study and it mainly examined FUSE Java Bindings performance [14]. With this paper we hope to increase the community discussion about the feasibility of moving file system and potentially the entire I/O stack development to the user space, as well as increase the effort for this migration.

4 User-Level File System Designs

General design options. Practical user-space file system design should not require application changes or recompilations. One can use *library preloading* to override default library calls, such as *open()* and *read()*. AVFS uses this method but its practicality is limited to the applications that access file system through shared libraries [3]. Another way is to have an in-kernel component that emulates an in-kernel file system but in reality communicates with a user-space daemon. Earlier works used distributed file system clients, such as NFS, Samba, and Coda, to avoid kernel modifications [12]. The user had to run a local user-space file system server that was modified to perform the required tasks, but this added unnecessary network stack overheads. So, creating an optimized in-kernel driver was the next logical option.

FUSE design. Although FUSE is generally associated with Linux, its ports exist in many modern OSes. High-level design is the same on all platforms (Figure 1). FUSE consists of an in-kernel driver and a multi-threaded user space daemon that interacts with the driver using a FUSE library. The driver registers itself with the Virtual File System (VFS) kernel layer as any other regular kernel-based file system. For the user it looks like another file system type supported by the kernel, but in fact it is a whole family of FUSE-based file systems.

Applications access FUSE file systems using regular I/O system calls. The VFS invokes the corresponding in-kernel file system—FUSE driver. The driver maintains an in-kernel request queue, which is read by the threads of the user-space daemon. The daemon implements the main file system logic. Often, the daemon has to call system calls to perform its tasks. E.g., for a write, a FUSE-based file system might need to write to an underlying file or send a network packet to a server. When request processing is completed, the daemon sends replies along with the data back to the driver.

Workload	Description
file-rread	Random read from a preallocated 30GB file.
file-rwrite	Random write to a preallocated 30GB file.
file-sread	Sequential read from a preallocated 60GB file.
file-swrite	Sequential write to a new 60GB file.
32files-sread	32 threads sequentially read 32 preallocated 1GB files. Each thread reads its own file.
32files-swrite	32 threads sequentially write 32 new 1GB files. Each thread writes its own file.
files-create	Create 4 million 4KB files in a flat directory.
files-read	Read 4 million 4KB files in a flat directory. Every next file to read is selected randomly.
files-delete	Deletion of 800,000 4KB preallocated files in a flat directory.
Web-server	Web-server workload emulated by Filebench. Scaled up to 1.25 million files.
Mail-server	Mail-server workload emulated by Filebench. Scaled up to 1.5 million files.
File-server	File-server workload emulated by Filebench. Scaled up to 200,000 files.

Table 1: Workload descriptions.

User-to-kernel and kernel-to-user communications cause the main overheads in FUSE. To reduce the number of such communications, current FUSE implementations typically support a cache. If a read comes from an application and the corresponding data was already previously read and cached in the kernel, then no communication with the user space is required. Starting from Linux 3.15 write-back cache is supported as well. Another common optimization implemented in FUSE is a zero-memory copying for moving the data. Figure 1 demonstrates that in a naïve implementation, the data often needs to cross the user-kernel boundary twice. To avoid that, advanced implementations use the *splice* functionality to copy or remap pages right in the kernel, without copying them to the user space.

5 Evaluation

We describe our experiments and results in this section.

Experimental setup. Performance degradation caused by FUSE depends heavily on the speed of underlying storage. To account for this we used three setups with different storage devices. The *HDD* setup used a Dell PowerEdge R710 machine with a 4-core Intel Xeon E5530 2.40GHz CPU. File systems were deployed on top of a Seagate Savvio 15K.2 disk drive (15KRPM, 146GB). The *Desktop SSD* setup used the same machine but file systems were deployed on Intel X25-M 200GB SSD. Finally, the *Enterprise SSD* setup

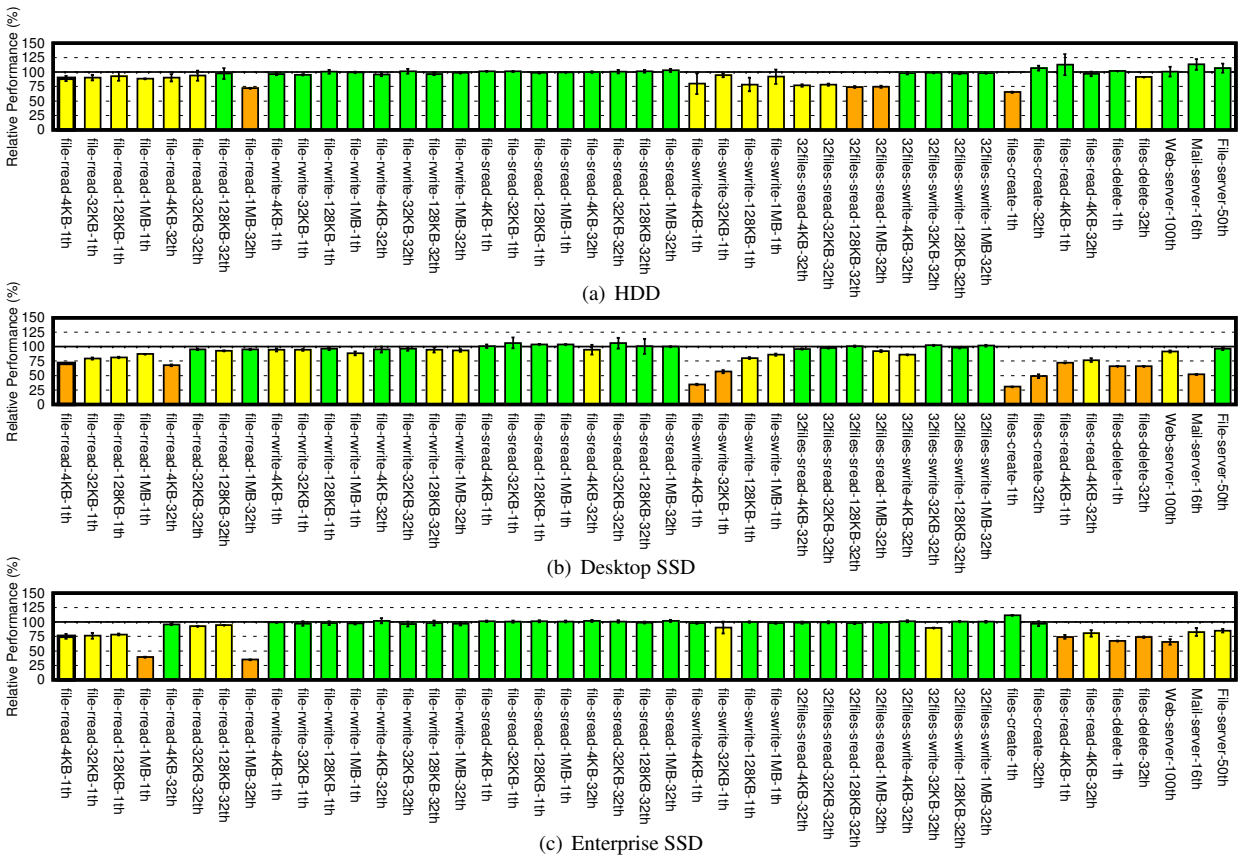

Figure 2: FUSE-Ext4 performance relative to Ext3 for a variety of workloads and devices.

was deployed on an IBM System x3650 M4 server equipped with two Intel Xeon CPUs (8 cores per CPU) and Micron MTFDDAK128MAR-1J SSD. We limited the amount of RAM available to the OS to 4GB in all setups to trigger accesses to non-cached data easier. The machines ran the CentOS 7 distribution, vanilla Linux 3.19, and libfuse commit 04ad73 dated the 2015-04-02.

We selected a set of micro workloads that covers a significant range of file system operations: sequential and random I/O, reads and writes, data- and meta-data-intensive workloads, single- and multi-threaded workloads, and used a variety of I/O sizes. All workloads were encoded using the Filebench Workload Model Language [6]. In addition, we ran three macro-workloads provided by Filebench: Web-, Mail-, and File-servers. Table 1 details our workload notations. All workloads, except files-create, files-read, files-delete, Web-, Mail-, and File-server were ran with 4KB, 32KB, 128KB, and 1MB I/O sizes. For files-read and files-create the I/O size was determined by the file size, 4KB. Web-, Mail-, and File-server are macro workloads for which I/O sizes and thread counts are dictated by Filebench. File-rread, file-rwrite, file-sread, files-create, files-read, and files-delete were executed for 1 and 32 threads. We did not run file-swrite for 32 threads, as such workload is less common in practice.

All experiments ran for at least 10 minutes and we present standard deviation on all graphs. To encourage reproducibility and further research, we published the details of the hardware and software configurations, all Filebench WML files, benchmarking scripts, and all raw experimental results here: `https://avatar.fsl.cs.sunysb.edu/groups/fuseperformancepublic/`.

FUSE. To estimate FUSE's overhead, we first measured the throughput of native Ext4. Then we measured the throughput of a FUSE *overlay* file system deployed on top of Ext4. The only functionality of the overlay file system is to pass requests to the underlying file system. We refer to this setup as *FUSE-Ext4*. Note that this setup implies that we measured higher FUSE overhead than one using a full-fledged user-space native Ext4 implementation (if it would exist).

We calculated the relative performance of FUSE-Ext4 compared to plain Ext4 for each workload. Figure 2 presents the results for three different storage devices. The relative performance varied across workloads and devices—from 31% (files-create-1th) to 100%. We categorized the workloads by FUSE's relative performance into four classes. (1) The *Green* class includes the workloads with relative performance higher than 95%. Here, there is almost no performance penalty and such workloads are ready to be serviced by user-space file systems.

(2) The *Yellow* class includes the workloads with 75% to 95% relative performance. Such overheads are often acceptable given the benefits that user-space development provides. (3) The *Orange* class consists of workloads that performed at 25–75% of Ext4. Such overheads might be acceptable only if user-space file systems provide major benefits. And finally, (4) the *Red* class includes workloads that demonstrated unacceptably low performance—less than 25% of an in-kernel file system.

Interestingly, none of the workloads fall into the red class. In our experiments with earlier FUSE versions (2 years earlier, not presented in this paper) up to 10% of workloads were in the red class. FUSE performance has clearly improved over time. Moreover, for all systems, most of the workloads are in the favorable, green class.

Differences in the relative performance of storage devices and CPUs cause some workloads to perform better on one system than another. For many write-intensive workloads we see that FUSE on the desktop SSD outperforms FUSE on the enterprise SSD. It turned out that our enterprise SSD is slower for writes than the desktop SSD. Our enterprise SSD maintains a constant write latency across its lifetime, which requires limiting the rate of writes. For the desktop SSD, however, writes are faster initially, but their speed is expected to degrade over time.

Results for macro workloads (Web-, File-, and Mail-server) are probably the most important as they are closer to real-world workloads. In the SSD setups performance degraded by 2–50% while for HDD setup we did not observe any performance penalty. In this position paper we do not perform detailed analysis of FUSE overhead. Instead, the results aim to intensify research in the overlooked area that has a profound impact on all aspects of storage design.

6 Conclusions

Modern file systems are complex software that are difficult to develop and maintain, especially in kernel space. We argue that a lot of file system development can be moved to user space in the future. Contrary to popular belief, our experiments demonstrated that for over 40 workloads, the throughput of user-space file systems has an acceptable level. Mature frameworks for writing user-space file systems are in place in many modern OSes, which opens the door for productive research. We believe that the research community should give more attention to evaluating and improving FUSE.

Acknowledgements. We thank Ravikant Malpani and Binesh Andrews for their help at the early stages of the project. We appreciate insightful comments from anonymous reviewers and our shepherd. This work was made possible in part thanks to NSF awards CNS-1302246, CNS-1305360, CNS-1522834, and IIS-1251137.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Technical Conference*, 1986.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. In *Proceedings of the Annual USENIX Technical Conference*, 1997.

[3] AVFS: A Virtual Filesystem. `http://avf.sourceforge.net/`.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: A checkpoint filesystem for parallel applications. Technical Report LA-UR 09-02117, LANL, April 2009. `http://institute.lanl.gov/plfs/`.

[5] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[6] Filebench. `http://filebench.sf.net`.

[7] H. Hartig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schonberg. The performance of Microkernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP '97)*, 1997.

[8] GNU Hurd. `www.gnu.org/software/hurd/hurd.html`.

[9] V. Kemerlis, G. Portokalidis, and A. Keromytis. kGuard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[10] The L4 Microkernel Family. `http://os.inf.tu-dresden.de/L4/`.

[11] Lessfs, January 2012. `www.lessfs.com`.

[12] P. Machek. UserVFS. `http://sourceforge.net/projects/uservfs/`.

[13] NTFS-3G. `www.tuxera.com`.

[14] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. In *25th Symposium On Applied Computing*, 2010.

[15] R. F. Rashid and G. G. Robertson. Accent: A communication oriented network operating system kernel. In *In Proc. 8th Symposium on Operating Systems Principles*, 1981.

[16] Opendedup, January 2012. `www.opendedup.org`.

[17] D. Steere, J. Kistler, and M. Satyanarayanan. Efficient user-level file cache management on the sun vnode interface. In *Proceedings of the Summer USENIX Technical Conference*, 1990.

[18] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net`, February 2005.

[19] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[20] A. Westerlund and J. Danielsson. Arla-a free AFS client. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, 1998.

[21] ZFS for Linux, January 2012. `www.zfs-fuse.net`.