Performance and Extension of User Space File Systems*

Aditya Rajgarhia
Stanford University
Stanford, CA 94305, USA
aditya@cs.stanford.edu

Ashish Gehani
SRI International
Menlo Park, CA 94025, USA
ashish.gehani@sri.com

ABSTRACT

Several efforts have been made over the years for developing file systems in user space. Many of these efforts have failed to make a significant impact as measured by their use in production systems. Recently, however, user space file systems have seen a strong resurgence. FUSE is a popular framework that allows file systems to be developed in user space while offering ease of use and flexibility.

In this paper, we discuss the evolution of user space file systems with an emphasis on FUSE, and measure its performance using a variety of test cases. We also discuss the feasibility of developing file systems in high-level programming languages, by using as an example Java bindings for FUSE that we have developed. Our benchmarks show that FUSE offers adequate performance for several kinds of workloads.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Language interconnection*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation*

Keywords

user space, FUSE, performance, Java, language binding

1. INTRODUCTION

Developing in-kernel file systems for Unix is a challenging task, due to a variety of reasons. This approach requires the programmer to understand and deal with complicated kernel code and data structures, making new code prone to bugs caused by programming errors. Moreover, there is a steep learning curve for doing kernel development due to the lack of facilities that are available to application programmers. For instance, the kernel code lacks memory protection, requires careful use of synchronization primitives, can be written only in C, and that too without being linked against the standard C library. Debugging kernel code is also tedious, and errors can require rebooting the system.

Even a fully functional in-kernel file system has several disadvantages. Porting a file system written for a particular flavor of Unix to a different one can require significant changes in the design and implementation of the file system, even though the use of similar file system interfaces (such as the VFS layer) on several Unix-like systems makes the task somewhat easier. Besides, an in-kernel file system can be mounted only with superuser privileges. This can be a hindrance for file system development and usage on centrally administered machines, such as those in universities and corporations.

In contrast to kernel development, programming in user space mitigates, or completely eliminates, several of the above-mentioned issues. Contemporary research in the area of storage and file systems increasingly involves the addition of rich functionality over a basic file system, as opposed to designing low-level file systems directly. As an example, the Ceph [37] distributed file system, designed to provide high performance, reliability and scalability, uses a client that is developed in user space. As we pointed out previously, developing extensive features in an in-kernel file system is not a trivial task. On the other hand, by developing in user space, the programmer has access to a wide range of familiar programming languages, third-party tools and libraries, and need not worry about the intricacies and challenges of kernel-level programming. Of course, user space file systems may still require some effort to port to different operating systems, depending on the extent to which a file system's implementation is coupled to a particular operating system's internals.

FUSE [34] is a widely used framework available for Unix-like operating systems, that allows nonprivileged users to develop file systems in user space. It has been merged into the Linux kernel, while ports are also available for several mainstream operating systems. FUSE requires a programmer to deal only with a simple application programming interface (API) consisting of familiar file system operations. A variety of programming language bindings are available for FUSE, which, coupled with its simplicity, makes file system development accessible to a large number of programmers. FUSE file systems can also be mounted by nonprivileged users, which motivates novel uses for file systems. For instance, WikipediaFS [2] allows a user to view and edit Wikipedia articles as if they were local files. SSHFS, which is distributed with the FUSE user space library, allows ac-

*This material is based upon work supported by the National Science Foundation under Grant OCI-0722068. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

cess to a remote file system via the SFTP protocol. Finally, developing a FUSE file system does not require recompilation or reboot of the kernel, adding to its convenience.

The prevailing view is that user space file systems suffer from significantly lower performance as compared to in-kernel implementations, due to overheads associated with additional context switches and memory copies. However, that may have changed due to increases in processor, memory, and bus speeds. Regular enhancements to the implementation of FUSE also contribute to augmented performance.

The remainder of the paper is organized as follows. Section 2 discusses previous work related to extending kernel functionality in user space. Section 3 gives an overview of FUSE, and explains the causes for its performance overhead. Section 4 describes the use of various programming languages for file system development, and our implementation of Java bindings for FUSE. Section 5 explains our benchmarking methodology, while the results of the benchmarks are presented in Section 6. Finally, Section 7 concludes.

2. BACKGROUND

Several efforts have been made to ease the development of file systems, some of them to facilitate the extensibility of operating systems in general.

Microkernel-based operating systems such as Mach [10] and Spring [11] typically implement only the most basic services such as address space management, process management, and interprocess communication within the kernel, while other functionality, including device drivers and file systems, is implemented as user space servers. However, the performance overhead of microkernels has been a contentious issue, and they have not been deployed widely. Other experimental approaches, such as the Exokernel [7] and L4 [21] overcome the performance issue, but these are still ongoing research efforts and their feasibility for widespread adoption is not clear.

Extensible operating systems such as Spin [1] and Vino [4] export interfaces to operating system services at varying degrees of granularity. The goal is to let applications safely modify system behavior at runtime, in order to achieve a particular level of performance and functionality. Like microkernels, extensible operating systems are still in the research domain.

Stackable file systems, first implemented in [28], allow new features to be added incrementally. Most notable of this approach is FiST [40], which allows file systems to be described using a high-level language. Using such a description, a code generator produces kernel file system modules for Linux, FreeBSD and Solaris without any modifications to the kernels. Maintaining flexibility while producing code for a variety of kernels, though, means that FiST file systems cannot easily manipulate low-level constructs such as the block layout of files on disks, and metadata structures such as inodes. Moreover, user space file systems have other advantages, such as the ability to be mounted by nonprivileged users without any intervention by the system administrator.

Using NFS loopback servers, portable user space file systems can be created. The toolkit described in [24] facilitates writing user space NFS loopback servers while avoiding several issues associated with this approach. Using the NFS interface can be beneficial for certain types of applications while affording portability. On the other hand, it means that file system semantics are limited due to NFS's weak cache consistency, and the POSIX file system semantics are often more suitable for file systems such as those dependent on low latency. Moreover, performance of NFS loopback servers is affected due to the overhead incurred in the network stack.

Various projects have aimed to support development of user space file systems while exporting an API similar to that of the VFS layer. UserFS [8] worked on Linux with kernel versions as high as 2.2. It consisted of a kernel module that registered a *userfs* file system type with the VFS. All requests to this file system were then communicated to a user space library through a file descriptor.

The Coda distributed file system [29] contains a user space cache manager, Venus. In order to service file system operations, the Coda kernel module communicates with Venus through a character device `/dev/cfs0`. UserVFS [22], which was developed as a replacement for UserFS, used this Coda character device for communication between the kernel module and the user space library. Thus, UserVFS did not require patching the kernel, since the Coda file system was already part of the Linux kernel. UserVFS was also ported to Solaris. Similarly, Arla [38] is an AFS client that consists of a kernel module, `xfs`, that communicates with the `arlad` user space daemon to serve file system requests.

The *ptrace()* system call can also be used to build an infrastructure for developing file systems in user space [31]. An advantage of this technique is that all OS entry points, rather than just file system operations, can be intercepted. The downside is that the overhead of using *ptrace()* is significant, which makes this approach unsuitable for production-level file systems.

Recent versions of NetBSD ship with puffs [15], which is similar to FUSE. The FUSE user space library interface has also been implemented on top of puffs, allowing the plethora of FUSE file systems available to run on NetBSD using the existing puffs kernel module [16].

Like some of the approaches mentioned above, FUSE consists of a loadable kernel module (although it can be compiled into the kernel as well) along with a user space library, `libfuse`. Unlike previous efforts, FUSE has been part of the mainline Linux kernel since version 2.6.14, and ports are also available for Mac OS X [30], OpenSolaris [9], FreeBSD [12] and NetBSD [16]. This reduces concerns that developers using FUSE may have about compatibility of their file systems with future versions of the kernel. Furthermore, FUSE is released under a flexible licensing scheme, enabling it to be used for free as well as commercial file systems. It is interesting to analyze the functionality and performance of FUSE since it is presently very widely used by researchers, corporations, the open source community and even hobbyists, as is evident from the following examples.

TierStore [6] is a distributed file system that aims to simplify the deployment of applications in network environments that lack the ability to support reliable, low-latency, end-to-end communication sessions, such as those in developing regions. TierStore uses either FUSE or an NFS loopback server to provide a file system interface to applications.

An increasing trend has been to install both Microsoft Windows as well as a flavor of Unix on the same machine. Historically, there has been a lack of support for accessing files stored in one operating system on the other due to incompatible file systems. NTFS-3G [25] is an open source implementation of the Microsoft Windows NTFS file system

with read/write support. Since NTFS-3G is implemented with FUSE, it can run unmodified on a variety of operating systems.

Likewise, ZFS-FUSE [41] is a FUSE implementation of the Sun Microsystems ZFS file system [32]. Porting ZFS to the Linux kernel is complicated by the fact that the GNU General Public License, which governs the Linux kernel, is incompatible with CDDL, the license under which ZFS is released. Since ZFS-FUSE runs in user space, though, it is not restricted by the kernel's license.

VMware's Disk Mount utility [36] allows a user to mount an unused virtual disk as a separate file system without the need to run a virtual machine. On Linux hosts, this is achieved by using FUSE.

3. FILE SYSTEM IN USER SPACE (FUSE)

We now describe the FUSE framework, and how a typical file system operation works in FUSE.

Once a FUSE volume is mounted, all file system calls targeting the mount point are forwarded to the FUSE kernel module, which registers a *fusefs* file system with the VFS. As an example, Figure 1 shows the call path through FUSE for a *read()* operation. The user space file system functionality is implemented as a set of callback functions in the `userfs` program, which is passed the mountpoint for the FUSE file system as a command line parameter. Assume the directory `/fuse` in the underlying file system is chosen as the FUSE mountpoint. When an application issues a *read()* system call for the file `/fuse/file`, the VFS invokes the appropriate handler in *fusefs*. If the requested data is found in the page cache, it is returned immediately. Otherwise, the system call is forwarded over a character device, `/dev/fuse`, to the `libfuse` library, which in turn invokes the callback defined in `userfs` for the *read()* operation. The callback may take any action, and return the desired data in the supplied buffer. For instance, it may do some pre-processing, request the data from the underlying file system (such as Ext3), and then post-process the read data. Finally, the result is propagated back by `libfuse`, through the kernel, and to the application that issued the *read()* system call. Other file system operations work in a similar manner.

A helper utility, `fusermount`, is provided to allow nonprivileged users to mount file systems. The `fusermount` utility allows several parameters to be customized at the time of mounting the file system.

FUSE provides a choice of two different APIs for developing the user space file system. The low-level API resembles the VFS interface closely, and the user space file system must manage the inodes and perform pathname translations (and any associated caching for the translations). Moreover, the file system must manually fill the data structure used for replying to the kernel. This interface is useful for file systems that are developed from scratch, such as ZFS-FUSE, as opposed to those that add functionality to an existing native file system.

On the other hand, the high-level API requires the file system developer to deal only with pathnames, rather than inodes, thus resembling system calls more closely than the VFS interface. `libfuse` is responsible for performing and caching the inode-to-pathname translations, and for filling and sending the reply data structures back to the kernel. The high-level API is sufficient for most purposes, and hence is used by the vast majority of FUSE-based file systems.

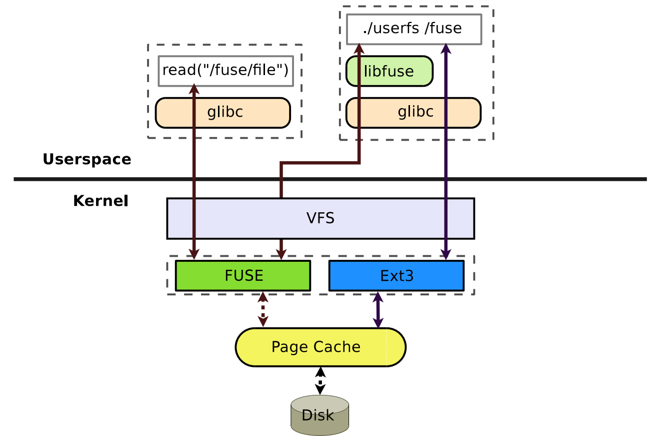


Figure 1: Typical file system call path through FUSE for a *read()* operation.

Performance Overhead of FUSE

When only a native file system such as Ext3 is used, there are two user-kernel mode switches per file system operation (to and from the kernel). No context switches need to be performed, where we use the term *context switch* to refer to the scheduler switching between processes with different address spaces. User-kernel mode switches are inexpensive and involve only switching the processor from unprivileged user mode to privileged kernel mode, or vice versa. However, using FUSE introduces two context switches for each file system call. There is a context switch from the user application that issued the system call to the FUSE user space library, and another one in the opposite direction. Therefore, when FUSE further passes operations to the underlying file system, a total of two context switches and two mode switches are performed. A context switch can have a significant cost, although the cost may depend vastly on a variety of factors such as the processor type, workload, and memory access patterns of the applications between which the context switch is performed. As discussed in [20, 5], the overhead of a context switch comes from several aspects - the processor registers need to be saved and restored, cache and translation lookaside buffer (TLB) entries for the evicted process need to be flushed and then reloaded for the incoming process, and the processor pipeline must be flushed.

FUSE also splits read and write requests into chunks of 128 KB, in order to minimize the number of dirty pages in the kernel that depend on the user space file system to be written to disk. If the kernel is under memory pressure due to a large number of such dirty pages, a deadlock may occur if the user space process that implements the responsible file system has been swapped out. Until recently, FUSE requests were split into chunks of only 4 KB, resulting in a significant number of context switches for each read and write operation that operated on buffers of size greater than 4 KB. The current 128 KB chunk size mitigates this issue since most applications, including common UNIX utilities such as `cp`, `cat`, `tar` and `sftp`, typically do not use buffers that are more than 32 KB. From Linux kernel 2.6.27 onward, using the `big_writes` mount option enables a 128 KB chunk size. Figure 2 shows the times required to write a 16 MB file using chunk sizes varying from 4 KB to 128 KB. As expected,

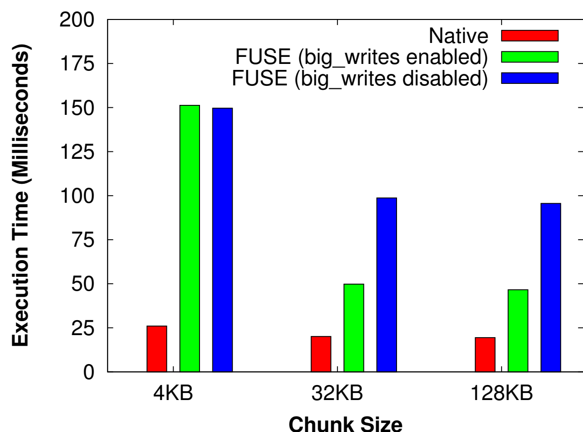


Figure 2: Time for writing a 16 MB file using various sizes for the *write()* buffer.

big_writes has no impact when the chunk size is 4 KB. However, for a chunk size of 32 KB, FUSE with big_writes enabled is approximately twice as fast as FUSE without big_writes. Note that the time without big_writes also decreases somewhat with an increase in chunk size, since fewer system calls (and hence context switches) need to be made. The time does not decrease when big_writes is used and the buffer size is increased from 32 KB to 128 KB. This may be due to the bottleneck being the memory bandwidth or the time required to set up the kernel data structures. Hence, 32 KB appears to be an ideal buffer size while using FUSE. Although the overhead of FUSE appears significant in Figure 2, it is only because a 16 MB file is small enough to fit entirely in the page cache. This is not a true reflection of FUSE's performance, as discussed further in Section 6.

The above-mentioned deadlock issue is most easily demonstrated through the use of shared writable mappings. With such mappings, the user space file system can create vast numbers of dirty pages without the kernel knowing about it. For this reason, FUSE did not support shared, writable mappings until Linux kernel 2.6.26, which added a dirty page accounting infrastructure that allows the number of pages used for write caching to be limited.

FUSE also introduces two additional memory copies, for arguments that are passed to or returned from the user space file system. Most significantly, these include the buffers containing data that is read or written. While using the native file system alone, data need be copied in memory only once, either from the kernel's page cache [1] to the application issuing the system call, or vice versa. While writing data to a FUSE file system, the data is first copied from the application to the page cache, then from the page cache to libfuse via /dev/fuse, and finally from libfuse to the page cache when the system call is made to the native file system. For *read()*, the copies are similarly performed in the other direction. Note that if the FUSE file system was mounted with the direct_io option, then the FUSE kernel module bypasses the page cache, and forwards the application-supplied buffer directly to the user space daemon. In this case, only one additional memory copy has to be performed. The advantage of using direct_io is that writes are significantly faster due to the reduced memory copy. The downside is that each *read()* request has to be forwarded to the user space file system, as data is not present in the page cache, thus affecting read performance severely. Nonetheless, the direct_io parameter is essential when only the file system can ensure the consistency of the data. As an example, if the data is read from physical sensors then content from the page cache should not be returned to the application.

Last, when using the native file system, all data that is read from or written to the disk is cached in the kernel's page cache. With FUSE, *fusefs* also caches the data in the page cache, resulting in two copies of the same data being cached. As we mentioned previously, if the data requested by an application is found in the page cache, it is returned immediately rather than the request being passed to libfuse. Although the use of the page cache by FUSE is very beneficial for read operations since it avoids unnecessary context switches and memory copies, the fact that the same data is cached twice reduces the efficiency of the page cache. In Linux, one can open files on the native file system using the O_DIRECT flag and thereby eliminate caching by the native file system. However, this is generally not a feasible solution since O_DIRECT imposes alignment restrictions on the length and address of the *write()* buffers and on the file offsets. Moreover, these restrictions vary by file system and kernel versions. As mentioned above, the direct_io mount option can be used to disable the page cache for FUSE, but the cost of doing so is far greater than the overhead of the double caching.

4. LANGUAGE BINDINGS FOR FUSE

More than twenty different language bindings are available for FUSE, allowing file systems to be written in languages other than C. This means that programmers can use languages that are based on different programming paradigms, offer different levels of type safety and type checking, and are generally intended for different usage scenarios. For instance, using a language such as C++ or C# allows the file system to be written in a high-performance, object-oriented language. On the other hand, functional languages like Haskell and OCaml provide tools such as higher-order functions and lazy evaluation that facilitate modularity and, in turn, productivity [13].

Some programming languages are particularly suitable for certain uses. For example, the Erlang programming language was designed to support fault-tolerant, real-time, non-stop applications in a distributed environment. As such, it has been used in several distributed storage systems [23, 26]. Halfs [14] is a file system developed using Haskell, which is well suited to high assurance development, a methodology for creating software systems that meet rigorously defined specifications with a high degree of confidence. The Ruby language, in which every data type is an object, was designed to encourage good interface design while providing a lot of flexibility to the programmer. Ruby is used in one project [35] to provide transparent access to online meteorological databases through a local file system interface. Outside the realm of file systems, the SPIN operating system [1] depends on key language features of Modula-3 to safely export fine-grained interfaces to operating system services.

[1]Starting from kernel verion 2.4.10, there is no separate buffer cache in Linux. The page cache stores both, pages resulting from accesses through the VM subsystem as well as those resulting from accesses through the VFS.

Several projects [33, 18] use the Python bindings for FUSE, since it allows for rapid software development. Python is an increasingly popular programming language, and has an extensive set of third-party libraries beyond a comprehensive standard library.

JavaFuse

We have implemented *JavaFuse* [27], a Java interface for the FUSE high-level API, by using the Java Native Interface (JNI) to communicate between the C and Java layers. Although a basic Java interface had already existed for FUSE, it is no longer maintained [19]. In addition, *JavaFuse* contains JNI code that allows the corresponding system call to be invoked from each Java callback method. This allows the file system developer to build functionality over the native file system without writing any JNI code.

JavaFuse allows file systems to be written in a combination of C and Java by utilizing the FUSE library, while aiming to provide maximum portability, flexibility, and ease of use. Since *JavaFuse* is a native executable, it inherits the UNIX runtime, which can be valuable for implementing file system features.

The developer writes the file system functionality as a Java class, and registers this class with *JavaFuse* using a command line parameter. For each FUSE callback, the Java class may have two types of methods that can communicate with *JavaFuse*: a *pre-call* and a *post-call*. The *pre-call* is passed the parameters that are received by the FUSE callback. Upon its completion, these parameters are copied back to *JavaFuse* and passed to the native system call. Finally, the parameters along with the result of the system call are passed to the *post-call*. Within a *pre-call* or *post-call* method, the file system developer is free to invoke any number of Java methods and use external libraries.

The behavior of *JavaFuse* can be customized for individual file systems by use of a configuration file. Each of the three types of calls (pre-calls, post-calls and system calls) can be switched on or off depending on the requirements of the file system, in order to reduce unnecessary overhead. Another important parameter that can be configured defines the behavior of the read and write callbacks. These operations may result in large amounts of data being copied between the C and Java layers. In certain file systems, copying the data may not be needed, in which case it can be avoided by specifying the *meta-data-only* parameter in the configuration file. An alternative technique would have been to use the *ByteBuffer* class provided by Java's Non-blocking I/O package, which would avoid memory copies between the Java virtual machine (JVM) and the native code. However, these buffers limit portability when compound data types such as C structures are used. Moreover, they are not safe for concurrent access, while FUSE uses multiple threads in order to augment performance.

5. BENCHMARK METHODOLOGY

Our goal is to determine the feasibility of developing and running file systems in user space. To this end, we have performed two kinds of benchmarks.

5.1 Microbenchmarks

Microbenchmarks were used to measure the performance of common file system operations and the raw throughput attainable. We have used a modified version of the Bonnie 2.0.6 benchmark tool [3]. Given a file size as a command line parameter, the benchmark proceeds in six phases. First, a file of the specified size is written using `putc()`, one character at a time. Then, the file is written from scratch again using 16 KB blocks. In steps three and four, the file is read one character at a time using `getc()`, and in 16 KB chunks using `read()`, respectively. Finally, these two read operations are performed again, this time after clearing the page cache before each step.

5.2 Macrobenchmarks

Although microbenchmarks are useful for determining the overhead of individual operations, application-level benchmarks provide an overall view of a file system's performance, and its viability for use in production systems. Hence, we have performed the following benchmarks that measure the file system performance for commonly used applications.

PostMark - This widely used benchmark was designed to replicate the small file workloads seen in electronic mail, Usenet news, and Web-based commerce under heavy load [17]. PostMark generates an initial pool of random text files whose range of sizes is configurable. Then, a specified number of transactions occurs on these files. PostMark is considered to be a good benchmark since it puts both file and metadata operations under heavy stress. We used version 1.51 of PostMark, with the configuration set to use 5,000 files ranging from 1 KB to 64 KB, and 50,000 transactions. This range of file sizes is representative of a typical Web server's workload [39].

Large File Copy - The time is measured for copying a 1.1 GB movie file using the `cp` command. This is very different from the PostMark benchmark since it involves a single, large file. Such large files are often used in desktop computers, multimedia servers, and scientific computing.

5.3 Tested Configurations

Comparisons were made between the following file system configurations:

Native - the underlying Ext4 file system.

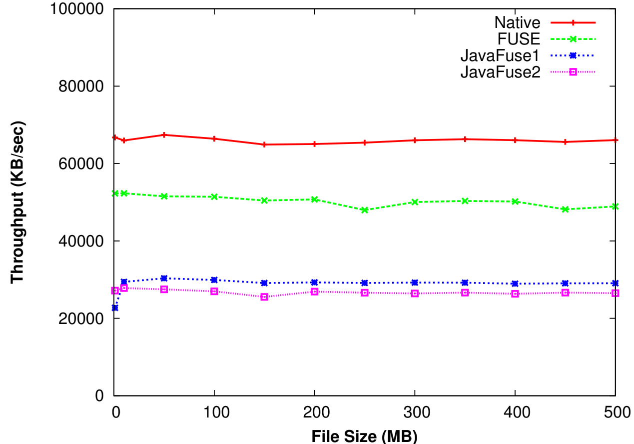
FUSE - A null FUSE file system written in C, which simply passes each call to the native file system.

JavaFuse1 (metadata-only) - A null file system written using *JavaFuse*, which does not copy the *read()* and *write()* buffers over the JNI call.

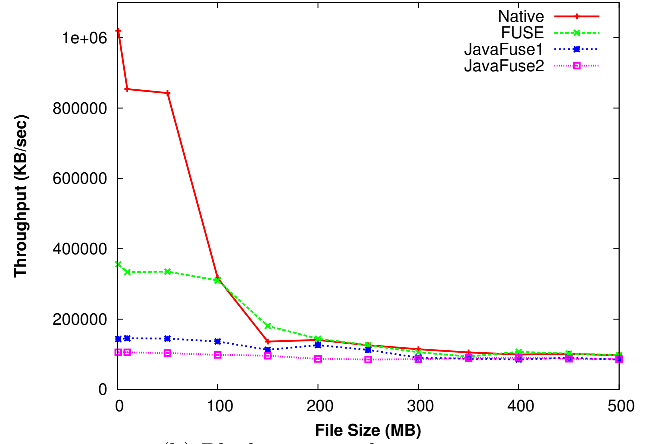
JavaFuse2 (copy all data) - Same as above, but also copies the *read()* and *write()* buffers over JNI.

5.4 Experimental Setup

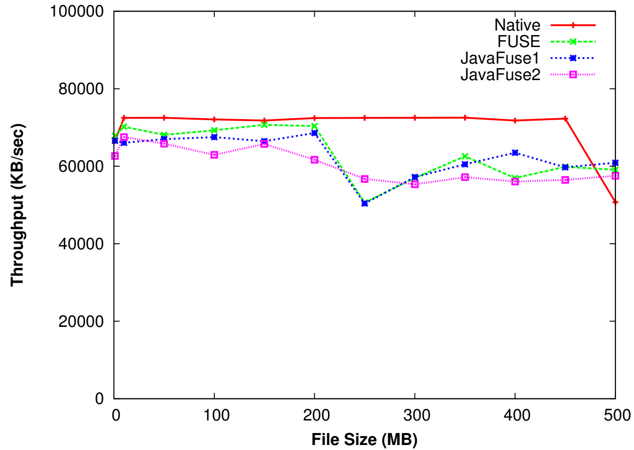
All benchmarks were performed on a single machine with a Pentium IV processor running at 3.40 GHz, 512 MB of main memory (which was increased to 2 GB for the macrobenchmarks, which generate a large number of files and hence require more memory to store the resulting file handles), and a 320 GB Seagate ST3320813AS SATA disk with 8 MB cache, running at 7,200 RPM. The maximum sustained transfer rate of this disk is 115 MB/s. Therefore, all observed throughputs that exceed 115 MB/s are benefiting either from the disk drive's cache or from the file system's page cache, and likely from a combination of both. The operating system was Linux, with kernel version 2.6.30.5, while the version of the user space FUSE library was 2.8.0-pre1. The native file system was Ext4.



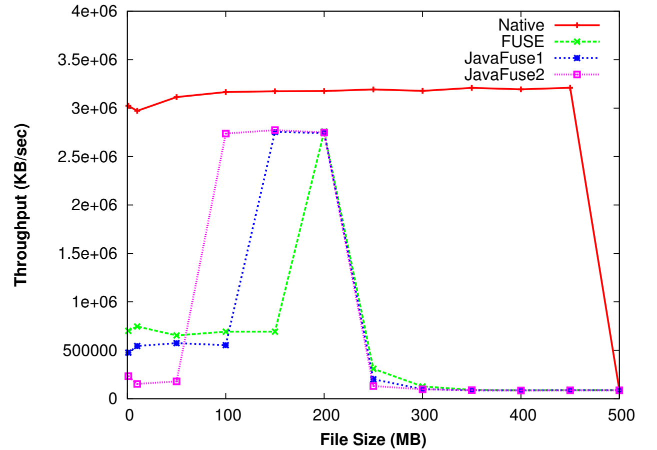
(a) Per-character sequential output.



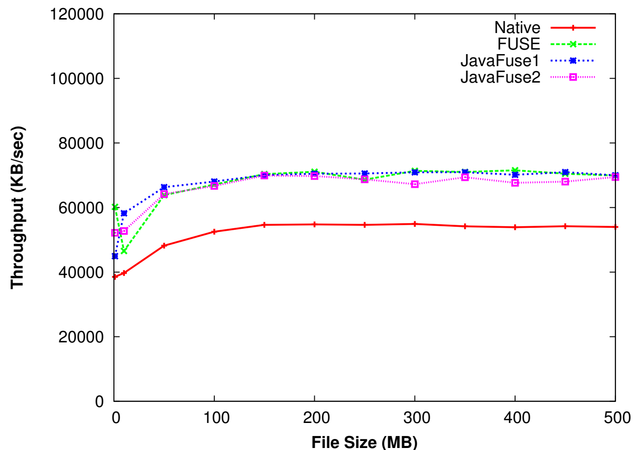
(b) Block sequential output.



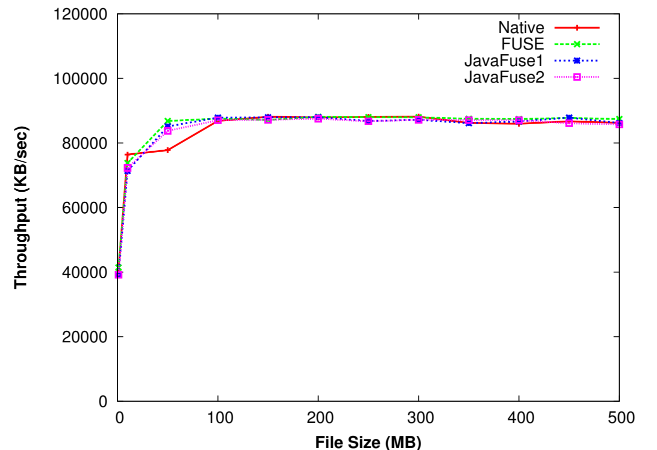
(c) Per-character sequential input.



(d) Block sequential input.



(e) Per-character sequential output (page cache cleared).



(f) Block sequential input (page cache cleared).

Figure 3: Microbenchmark results.

6. DISCUSSION

The results of the benchmarks are analyzed below. All benchmarks were performed five times, and the average value was used. We cleared the page cache before running each benchmark. The following discussion assumes an understanding of how FUSE works, as described in Section 3.

6.1 Microbenchmark Results

Figure 3 presents the output from the microbenchmarks. For the per-character sequential output in Figure 3(a), FUSE has approximately 25% overhead, caused by the large number of additional context switches. Likewise, using *JavaFuse* degrades performance even further due to another layer of context switches between `libfuse` and the JVM. For block sequential output (Figure 3(b)), Native is much faster for file sizes less than 100 MB. After that, the throughputs for Native and FUSE converge because for smaller file sizes, data is written only to the page cache, and not to the disk. Writing to the page cache does not consume much time, and hence the overhead of FUSE appears relatively large. On the other hand, once the file size is big enough to force data to be written to the disk, the I/O time dominates, and hence there is not much relative difference between the Native and FUSE throughputs.

Since the benchmark reads the same file that it wrote in the preceding step, part of the file now resides in the page cache. As we had explained earlier, data is cached twice when FUSE is used. For per-character input (Figure 3(c)), FUSE has less than 5% overhead for file sizes as large as 200 MB. After this point, the file can no longer be accommodated completely in the page cache when accessed through FUSE, and it's throughput drops slightly. Similar to the case of per-character input, files as large as 450 MB are stored completely in the page cache in the case of Native for the block input test (Figure 3(d)). For both FUSE and *JavaFuse*, the block input test contains a spike in the throughput due to caching effects caused by the synthetic nature of the benchmark; in the previous step (per-character sequential input), when the file can no longer fit in the page cache, the initial part is written out to disk. However, the latter part of the file still resides in the page cache, and is thus accessed directly in the current step, causing the spike.

To negate the effects of the test file already residing in the page cache, the previous two steps are performed again with empty page caches. Because the page cache was cleared before this step, the throughputs start out slow for both native and FUSE in the per-character input test (Figure 3(e)). The kernel determines that the file is being read sequentially, and enables read-ahead caching in order to improve performance. Surprisingly, FUSE obtains a higher sustained throughput than Native. The 2.6 series of the Linux kernel employs a complex read-ahead mechanism, and we believe that better use of this mechanism is being made when FUSE is used. This may be due to read-ahead being enabled for both, *fusefs* as well as Ext4, when a file is accessed through FUSE. When block input is performed (Figure 3(f)), Native and FUSE seem to benefit equally from the read-ahead logic, and there is no visible overhead for FUSE regardless of the file size.

6.2 Macrobenchmark Results

The native ext3 file system completes the PostMark benchmark in 84 seconds, while FUSE requires 91 seconds (see Figure 4). Thus, FUSE incurs an overhead of less than 10%. With Java, however, the overhead (more than 60%) is much greater due to higher CPU and memory usage. Although the benchmark is synthetic and does not by itself imply that FUSE can be used for production servers, it does put heavy stress on the file system, and is certainly an indication of the performance that can be obtained by FUSE.

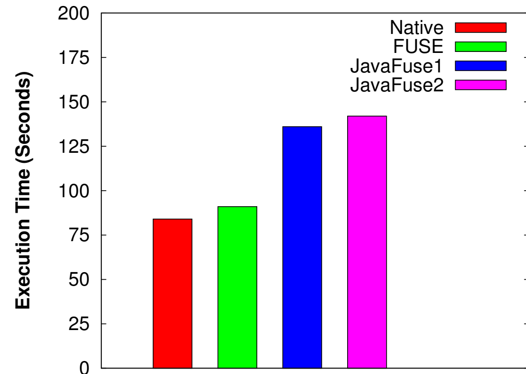


Figure 4: Time for executing PostMark.

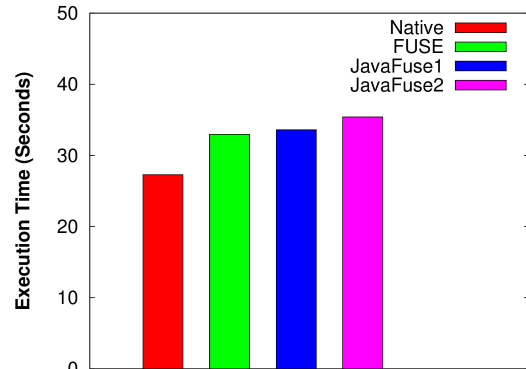


Figure 5: Time for copying a 1.1 GB video file.

As seen in Figure 5, FUSE also performs comparably to Native while copying a 1.1 GB file. This corroborates the results of the block input throughputs from the microbenchmarks, where the overhead of FUSE was negligible when the cost of performing disk I/O was the dominating factor.

7. CONCLUSION

Our benchmarks show that whether FUSE is a feasible solution depends on the expected workload for a system. It achieves performance comparable to in-kernel file systems when large, sustained I/O is performed. On the other hand, the overhead of FUSE is more visible when the workload consists of a relatively large number of metadata operations, such as those seen in Web servers and other systems that deal with small files and a very large number of clients. FUSE is certainly an adequate solution for personal computers and small-scale servers, especially those that perform large I/O transfers, such as Grid applications and multimedia servers.

Using an additional programming language layer in the form of *JavaFuse* incurred visible overhead for CPU intensive benchmarks due to the presence of the JVM. However, with proper optimization, the overhead may be reduced. For instance, if the language is compiled to native code and shared buffers are used, then the overhead should be negligible.

8. REFERENCES

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, S. Eggers, Extensibility, safety and performance in the SPIN operating system, 15th ACM Symposium on Operating System Principles (SOSP), 1995.

[2] M. Blondel, WikipediaFS, http://wikipediafs.sourceforge.net

[3] T. Bray, Bonnie, http://www.textuality.com/bonnie

[4] D. R. Cheriton, and K. J. Duda, A caching model of operating system kernel functionality, 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI), 1994.

[5] F. M. David, J. C. Carlyle, and R. H. Campbell, Context switch overheads for Linux on ARM platforms, Workshop on Experimental Computer Science, 2007.

[6] H. Demmer, B. Du, and E. Brewer, TierStore: a distributed file system for challenged networks in developing regions, 6th USENIX Conference on File and Storage Technologies (FAST), 2008.

[7] D. R. Engler, M. Frans Kaashoek, and J. O'Toole Jr., Exokernel: an operating system architecture for application-level resource management, 15th ACM Symposium on Operating System Principles (SOSP), 1995.

[8] J. Fitzhardinge, UserFS, http://www.goop.org/~jeremy/userfs

[9] FUSE on OpenSolaris, http://opensolaris.org/os/project/fuse

[10] D. Golub, R. Dean, A. Forin, and R. Rashid, Unix as an application program, Summer USENIX Technical Conference, 1990.

[11] G. Hamilton and P. Kougiouris, The Spring nucleus: A microkernel for objects, Summer USENIX Technical Conference, 1993.

[12] C. Henk, FreeBSD FUSE, http://fuse4bsd.creo.hu

[13] J. Hughes, Why functional programming matters, The Computer Journal, 1989.

[14] I. Jones, Halfs, ACM SIGPLAN Workshop on Haskell, 2005.

[15] A. Kantee, puffs - Pass-to-userspace framework file system, AsiaBSDCon, 2007.

[16] A. Kantee, ReFUSE: Userspace FUSE reimplementation using puffs, EuroBSDCon, 2007.

[17] J. Katcher, PostMark: A new file system benchmark, Technical Report TR3022, Network Appliance, 1997.

[18] D. Letscher, Carbon copy file system: A real-time snapshot file system layer, http://dehn.slu.edu/research/papers/usenix06.pdf

[19] P. Levart, FUSE-J, http://sourceforge.net/projects/fuse-j

[20] C. Li, C. Ding, and K. Shen, Quantifying the cost of context switch, ACM Workshop on Experimental Computer Science, 2007.

[21] J. Liedtke, On micro-kernel construction, ACM SIGOPS Operating Systems Review, Vol. 29(5), 1995.

[22] P. Machek, UserVFS, http://sourceforge.net/projects/uservfs

[23] H. Mattsson, H. Nilsson, and C. Wikstrom, Mnesia - A distributed robust DBMS for telecommunications applications, Practical Applications of Declarative Languages Symposium, 1999.

[24] D. Mazieres, A toolkit for user-level file systems, Usenix Technical Conference, 2001.

[25] NTFS-3G, http://www.ntfs-3g.com

[26] J. Paris, M. Gulias, A. Valderruten, and S. Jorge, A distributed file system for spare storage, Springer-Verlag Lecture Notes in Computer Science, Vol. 4739, 2007.

[27] A. Rajgarhia, JavaFuse, http://sourceforge.net/projects/javafuse

[28] D. S. Rosenthal, Evolving the vnode interface, Summer USENIX Technical Conference, 1990.

[29] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, Coda: A highly available file system for a distributed workstation environment, IEEE Transaction on Computers, Vol. 39(4), 1990.

[30] A. Singh, MacFUSE, http://code.google.com/p/macfuse

[31] R. P. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok, Rapid file system development using ptrace, ACM Workshop on Experimental Computer Science (EXPCS), 2007.

[32] ZFS, Sun Microsystems, http://www.opensolaris.org/os/community/zfs

[33] M. Slawinska, J. Slawinski, and V. Sunderam, Enhancing build-portability for scientific applications across heterogeneous platforms, IEEE International Symposium on Parallel and Distributed Processing, 2008.

[34] M. Szeredi, File system in user space, http://fuse.sourceforge.net

[35] K. Ui, T. Amagasa, and H. Kitagawa, A FUSE-based tool for accessing meteorological data in remote servers, 20th International Conference on Scientific and Statistical Database Management, 2008.

[36] VMware Disk Mount User's Guide, http://www.vmware.com/pdf/VMwareDiskMount.pdf

[37] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, Ceph: A scalable, high-performance distributed file system, 7th Symposium on Operating Systems Design and Implementation (OSDI), 2000.

[38] A. Westerlund, and J. Danielsson, Arla - a free AFS client, 1998 USENIX Technical Conference, Freenix track, 1998.

[39] A. Williams, M. Arlitt, C. Williamson, and K. Barker, Web workload characterization: Ten years later, Web Information Systems Engineering and Internet Technologies Book Series, Vol. 2, 2006.

[40] E. Zadok, and J. Nieh, FiST: A language for stackable file systems, Usenix Technical Conference, 2000.

[41] ZFS-FUSE, http://www.wizy.org/wiki/ZFS_on_FUSE