To FUSE or Not to FUSE: Performance of User-Space File Systems

Bharath Kumar Reddy Vangoor, ***Stony Brook University;*** **Vasily Tarasov,** ***IBM Research-Almaden;*** **Erez Zadok,** ***Stony Brook University***

https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Open access to the Proceedings of the 15th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

To FUSE or Not to FUSE: Performance of User-Space File Systems

Bharath Kumar Reddy Vangoor[1], Vasily Tarasov[2], and Erez Zadok[1]
[1]*Stony Brook University* *and* [2]*IBM Research—Almaden*

Abstract

Traditionally, file systems were implemented as part of OS kernels. However, as complexity of file systems grew, many new file systems began being developed in user space. Nowadays, user-space file systems are often used to prototype and evaluate new approaches to file system design. Low performance is considered the main disadvantage of user-space file systems but the extent of this problem has never been explored systematically. As a result, the topic of user-space file systems remains rather controversial: while some consider user-space file systems a toy not to be used in production, others develop full-fledged production file systems in user space. In this paper we analyze the design and implementation of the most widely known user-space file system framework—FUSE—and characterize its performance for a wide range of workloads. We instrumented FUSE to extract useful statistics and traces, which helped us analyze its performance bottlenecks and present our analysis results. Our experiments indicate that depending on the workload and hardware used, performance degradation caused by FUSE can be completely imperceptible or as high as –83% even when optimized; and relative CPU utilization can increase by 31%.

1 Introduction

File systems offer a common interface for applications to access data. Although micro-kernels implement file systems in user space [1, 16], most file systems are part of monolithic kernels [6, 22, 34]. Kernel implementations avoid the high message-passing overheads of micro-kernels and user-space daemons [7, 14].

In recent years, however, user-space file systems rose in popularity for four reasons. (1) Several stackable file systems add specialized functionality over existing file systems (e.g., deduplication and compression [19, 31]). (2) In academia and R&D settings, this framework enabled quick experimentation and prototyping of new approaches [3, 9, 15, 21, 40]. (3) Several existing kernel-level file systems were ported to user space (e.g., ZFS [45], NTFS [25]). (4) More companies rely on user-space implementations: IBM'S GPFS [30] and LTFS [26], Nimble Storage's CASL [24], Apache's HDFS [2], Google File System [13], RedHat's GlusterFS [29], Data Domain's DDFS [46], etc.

Increased file systems complexity is a contributing factor to user-space file systems' growing popularity (e.g., Btrfs is over 85 KLoC). User space code is easier to develop, port, and maintain. Kernel bugs can crash whole systems, whereas user-space bugs' impact is more contained. Many libraries and programming languages are available in user-space in multiple platforms.

Although user-space file systems are not expected to displace kernel file systems entirely, they undoubtedly occupy a growing niche, as some of the more heated debates between proponents and opponents indicate [20, 39, 41]. The debates center around two trade-off factors: (1) how large is the performance overhead caused by a user-space implementations and (2) how much easier is it to develop in user space. Ease of development is highly subjective, hard to formalize and therefore evaluate; but performance is easier to evaluate empirically. Oddly, little has been published on the performance of user-space file system frameworks.

In this paper we use a popular user-space file system framework, *FUSE*, and characterize its performance. We start with a detailed explanation of FUSE's design and implementation for four reasons: (1) the architecture is somewhat complex; (2) little information on internals is available publicly; (3) FUSE's source code can be difficult to analyze, with complex asynchrony and user-kernel communications; and (4) as FUSE's popularity grows, a detailed analysis of its implementation becomes of high value to many.

We developed a simple pass-through stackable file system in FUSE and then evaluated its performance when layered on top of Ext4 compared to native Ext4. We used a wide variety of micro- and macro-workloads, and different hardware using basic and optimized configurations of FUSE. We found that depending on the workload and hardware, FUSE can perform as well as Ext4, but in the worst cases can be 3× slower. Next, we designed and built a rich instrumentation system for FUSE to gather detailed performance metrics. The statistics extracted are applicable to any FUSE-based systems. We then used this instrumentation to identify bottlenecks in FUSE, and to explain why, for example, its performance varied greatly for different workloads.

2 FUSE Design

FUSE—Filesystem in Userspace—is the most widely used user-space file system framework [35]. According to the most modest estimates, at least 100 FUSE-based file systems are readily available on the Web [36]. Although other, specialized implementations of user-space file systems exist [30, 32, 42], we selected FUSE for this study because of its high popularity.

Although many file systems were implemented using

FUSE—thanks mainly to the simple API it provides—little work was done on understanding its internal architecture, implementation, and performance [27]. For our evaluation it was essential to understand not only FUSE's high-level design but also some details of its implementation. In this section we first describe FUSE's basics and then we explain certain important implementation details. FUSE is available for several OSes: we selected Linux due to its wide-spread use. We analyzed the code of and ran experiments on the latest stable version of the Linux kernel available at the beginning of the project—v4.1.13. We also used FUSE library commit #386b1b; on top of FUSE v2.9.4, this commit contains several important patches which we did not want exclude from our evaluation. We manually examined all new commits up to the time of this writing and confirmed that no new major features or improvements were added to FUSE since the release of the selected versions.

2.1 High-Level Architecture

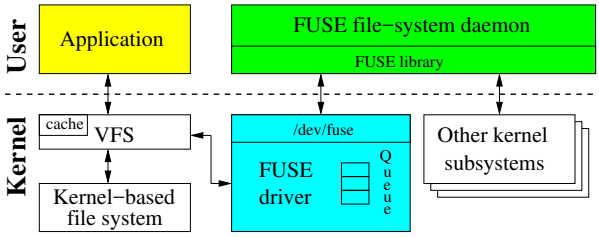

Figure 1: FUSE high-level architecture.

FUSE consists of a kernel part and a user-level daemon. The kernel part is implemented as a Linux kernel module that, when loaded, registers a *fuse* file-system driver with Linux's VFS. This *Fuse* driver acts as a proxy for various specific file systems implemented by different user-level daemons.

In addition to registering a new file system, FUSE's kernel module also registers a `/dev/fuse` block device. This device serves as an interface between user-space FUSE daemons and the kernel. In general, daemon reads FUSE requests from `/dev/fuse`, processes them, and then writes replies back to `/dev/fuse`.

Figure 1 shows FUSE's high-level architecture. When a user application performs some operation on a mounted FUSE file system, the VFS routes the operation to FUSE's kernel driver. The driver allocates a FUSE request structure and puts it in a FUSE queue. At this point, the process that submitted the operation is usually put in a wait state. FUSE's user-level daemon then picks the request from the kernel queue by reading from `/dev/fuse` and processes the request. Processing the request might require re-entering the kernel again: for example, in case of a stackable FUSE file system, the daemon submits operations to the underlying file system (e.g., Ext4); or in case of a block-based FUSE file system, the daemon reads or writes from the block device. When done with processing the request, the FUSE daemon writes the response back to `/dev/fuse`; FUSE's kernel driver then marks the request as completed and wakes up the original user process.

Some file system operations invoked by an application can complete without communicating with the user-level FUSE daemon. For example, reads from a file whose pages are cached in the kernel page cache, do not need to be forwarded to the FUSE driver.

2.2 Implementation Details

We now discuss several important FUSE implementation details: the user-kernel protocol, library and API levels, in-kernel FUSE queues, splicing, multi-threading, and write-back cache.

Group (#)	Request Types
Special (3)	**INIT**, **DESTROY**, **INTERRUPT**
Metadata (14)	**LOOKUP**, **FORGET**, **BATCH_FORGET**, CREATE, UNLINK, LINK, RENAME, RENAME2, **OPEN**, **RELEASE**, STATFS, FSYNC, **FLUSH**, **ACCESS**
Data (2)	READ, WRITE
Attributes (2)	GETATTR, SETATTR
Extended Attributes (4)	SETXATTR, GETXATTR, LISTXATTR, REMOVEXATTR
Symlinks (2)	SYMLINK, READLINK
Directory (7)	MKDIR, RMDIR, **OPENDIR**, **RELEASEDIR**, READDIR, **READDIRPLUS**, FSYNCDIR
Locking (3)	GETLK, SETLK, SETLKW
Misc (6)	BMAP, FALLOCATE, MKNOD, IOCTL, POLL, NOTIFY_REPLY

Table 1: FUSE request types, by group (whose size is in parenthesis). Requests we discuss in the text are in bold.

User-kernel protocol. When FUSE's kernel driver communicates to the user-space daemon, it forms a *FUSE request* structure. Requests have different types depending on the operation they convey. Table 1 lists all 43 FUSE request types, grouped by their semantics. As seen, most requests have a direct mapping to traditional VFS operations: we omit discussion of obvious requests (e.g., READ, CREATE) and instead next focus on those less intuitive request types (marked bold in Table 1).

The INIT request is produced by the kernel when a file system is mounted. At this point user space and kernel negotiate (1) the protocol version they will operate on (7.23 at the time of this writing), (2) the set of mutually supported capabilities (e.g., READDIRPLUS or FLOCK support), and (3) various parameter settings (e.g., FUSE read-ahead size, time granularity). Conversely, the DESTROY request is sent by the kernel during the file system's unmounting process. When getting a DESTROY, the daemon is expected to perform all necessary

cleanups. No more requests will come from the kernel for this session and subsequent reads from `/dev/fuse` will return 0, causing the daemon to exit gracefully.

The INTERRUPT request is emitted by the kernel if any previously sent requests are no longer needed (e.g., when a user process blocked on a READ is terminated). Each request has a unique *sequence#* which INTERRUPT uses to identify victim requests. Sequence numbers are assigned by the kernel and are also used to locate completed requests when the user space replies.

Every request also contains a *node ID*—an unsigned 64-bit integer identifying the inode both in kernel and user spaces. The path-to-inode translation is performed by the LOOKUP request. Every time an existing inode is looked up (or a new one is created), the kernel keeps the inode in the inode cache. When removing an inode from the dcache, the kernel passes the FORGET request to the user-space daemon. At this point the daemon might decide to deallocate any corresponding data structures. BATCH_FORGET allows kernel to forget multiple inodes with a single request.

An OPEN request is generated, not surprisingly, when a user application opens a file. When replying to this request, a FUSE daemon has a chance to optionally assign a 64-bit *file handle* to the opened file. This file handle is then returned by the kernel along with every request associated with the opened file. The user-space daemon can use the handle to store per-opened-file information. E.g., a stackable file system can store the descriptor of the file opened in the underlying file system as part of FUSE's file handle. FLUSH is generated every time an opened file is closed; and RELEASE is sent when there are no more references to a previously opened file.

OPENDIR and RELEASEDIR requests have the same semantics as OPEN and RELEASE, respectively, but for directories. The READDIRPLUS request returns one or more directory entries like READDIR, but it also includes metadata information for each entry. This allows the kernel to pre-fill its inode cache (similar to NFSv3's READDIRPLUS procedure [4]).

When the kernel evaluates if a user process has permissions to access a file, it generates an ACCESS request. By handling this request, the FUSE daemon can implement custom permission logic. However, typically users mount FUSE with the *default_permissions* option that allows kernel to grant or deny access to a file based on its standard Unix attributes (ownership and permission bits). In this case no ACCESS requests are generated.

Library and API levels. Conceptually, the FUSE library consists of two levels. The lower level takes care of (1) receiving and parsing requests from the kernel, (2) sending properly formatted replies, (3) facilitating file system configuration and mounting, and (4) hiding potential version differences between kernel and user space. This part exports the *low-level FUSE API*.

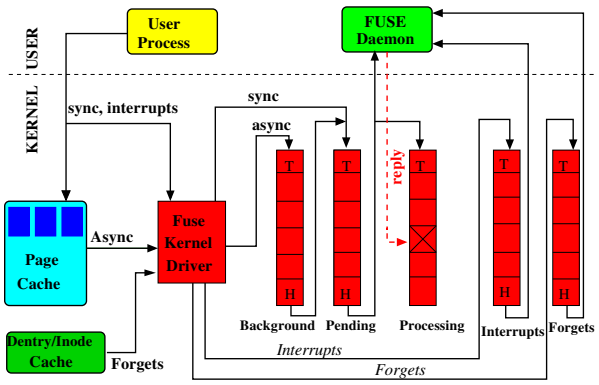

Figure 2: The organization of FUSE queues marked with their Head and Tail. The processing queue does not have a tail because the daemon replies in an arbitrary order.

The *High-level FUSE API* builds on top of the low-level API and allows developers to skip the implementation of the *path-to-inode* mapping. Therefore, neither inodes nor lookup operations exist in the high-level API, easing the code development. Instead, all high-level API methods operate directly on file paths. The high-level API also handles request interrupts and provides other convenient features: e.g., developers can use the more common `chown()`, `chmod()`, and `truncate()` methods, instead of the lower-level `setattr()`. File system developers must decide which API to use, by balancing flexibility vs. development ease.

Queues. In Section 2.1 we mentioned that FUSE's kernel has a request queue. FUSE actually maintains five queues as seen in Figure 2: (1) *interrupts*, (2) *forgets*, (3) *pending*, (4) *processing*, and (5) *background*. A request belongs to only one queue at any time. FUSE puts INTERRUPT requests in the interrupts queue, FORGET requests in the forgets queue, and synchronous requests (e.g., metadata) in the pending queue. When a file-system daemon reads from `/dev/fuse`, requests are transferred to the user daemon as follows: (1) Priority is given to requests in the interrupts queue; they are transferred to the user space before any other request. (2) FORGET and non-FORGET requests are selected fairly: for each 8 non-FORGET requests, 16 FORGET requests are transferred. This reduces the burstiness of FORGET requests, while allowing other requests to proceed. The oldest request in the pending queue is transferred to the user space and simultaneously moved to the processing queue. Thus, processing queue requests are currently processed by the daemon. If the pending queue is empty then the FUSE daemon is blocked on the read call. When the daemon replies to a request (by writing to `/dev/fuse`), the corresponding request is removed from the processing queue.

The background queue is for staging asynchronous requests. In a typical setup, only read requests go to

the background queue; writes go to the background queue too but only if the writeback cache is enabled. In such configurations, writes from the user processes are first accumulated in the page cache and later `bdflush` threads wake up to flush dirty pages [8]. While flushing the pages FUSE forms asynchronous write requests and puts them in the background queue. Requests from the background queue gradually trickle to the pending queue. FUSE limits the number of asynchronous requests simultaneously residing in the pending queue to the configurable `max_background` parameter (12 by default). When fewer than 12 asynchronous requests are in the pending queue, requests from the background queue are moved to the pending queue. The intention is to limit the delay caused to important synchronous requests by bursts of background requests.

The queues' lengths are not explicitly limited. However, when the number of asynchronous requests in the pending and processing queues reaches the value of the tunable `congestion_threshold` parameter (75% of `max_background`, 9 by default), FUSE informs the Linux VFS that it is congested; the VFS then throttles the user processes that write to this file system.

Splicing and FUSE buffers. In its basic setup, the FUSE daemon has to `read()` requests from and `write()` replies to `/dev/fuse`. Every such call requires a memory copy between the kernel and user space. It is especially harmful for WRITE requests and READ replies because they often process a lot of data. To alleviate this problem, FUSE can use *splicing* functionality provided by the Linux kernel [38]. Splicing allows the user space to transfer data between two in-kernel memory buffers without copying the data to user space. This is useful, e.g., for stackable file systems that pass data directly to the underlying file system.

To seamlessly support splicing, FUSE represents its buffers in one of two forms: (1) a regular memory region identified by a pointer in the user daemon's address space, or (2) a kernel-space memory pointed by a file descriptor. If a user-space file system implements the `write_buf()` method, then FUSE splices the data from `/dev/fuse` and passes the data directly to this method in a form of the buffer containing a file descriptor. FUSE splices WRITE requests that contain more than a single page of data. Similar logic applies to replies to READ requests with more than two pages of data.

Multithreading. FUSE added multithreading support as parallelism got more popular. In multi-threaded mode, FUSE's daemon starts with one thread. If there are two or more requests available in the pending queue, FUSE automatically spawns additional threads. Every thread processes one request at a time. After processing the request, each thread checks if there are more than 10 threads; if so, that thread exits. There is no explicit upper limit on the number of threads created by the FUSE library. An implicit limit exists for two reasons: (1) by default, only 12 asynchronous requests (`max_background` parameter) can be in the pending queue at one time; and (2) the number of synchronous requests in the pending queue depends on the total amount of I/O activity generated by user processes. In addition, for every INTERRUPT and FORGET requests, a new thread is invoked. In a typical system where there is no interrupts support and few FORGETs are generated, the total number of FUSE daemon threads is at most $(12 + number\ of\ requests\ in\ pending\ queue)$.

Write back cache and max writes. The basic write behavior of FUSE is synchronous and only 4KB of data is sent to the user daemon for writing. This results in performance problems on certain workloads; when copying a large file into a FUSE file system, `/bin/cp` indirectly causes every 4KB of data to be sent to userspace synchronously. The solution FUSE implemented was to make FUSE's page cache support a write-back policy and then make writes asynchronous. With that change, file data can be pushed to the user daemon in larger chunks of `max_write` size (limited to 32 pages).

3 Instrumentation

To study FUSE's performance, we developed a simple stackable passthrough file system—called *Stackfs*—and instrumented FUSE's kernel module and user-space library to collect useful statistics and traces. We believe that the instrumentation presented here is useful for anyone who develops a FUSE-based file system.

3.1 Stackfs

Stackfs is a file system that passes FUSE requests unmodified directly to the underlying file system. The reason for Stackfs was twofold. (1) After examining the code of all publicly available [28, 43] FUSE-based file systems, we found that most of them are stackable (i.e., deployed on top of other, often in-kernel file systems). (2) We wanted to add as little overhead as possible, to isolate the overhead of FUSE's kernel and library.

Complex production file systems often need a high degree of flexibility, and thus use FUSE's low-level API. As such file systems are our primary focus, we implemented Stackfs using FUSE's low-level API. This also avoided the overheads added by the high-level API. Below we describe several important data structures and procedures that Stackfs uses.

Inode. Stackfs stores per-file metadata in an inode. Stackfs's inode is not persistent and exists in memory only while the file system is mounted. Apart from bookkeeping information, the inode stores the path to the underlying file, its inode number, and a reference counter.

The path is used, e.g., to open the underlying file when an OPEN request for a Stackfs file arrives.

Lookup. During lookup, Stackfs uses `stat(2)` to check if the underlying file exists. Every time a file is found, Stackfs allocates a new inode and returns the required information to the kernel. Stackfs assigns its inode the number equal to the address of the inode structure in memory (by typecasting), which is guaranteed to be unique. This allows Stackfs to quickly find the inode structure for any operations following the lookup (e.g., open or stat). The same inode can be looked up several times (e.g., due to hardlinks) and therefore Stackfs stores inodes in a hash table indexed by the underlying inode number. When handling LOOKUP, Stackfs checks the hash table to see whether the inode was previously allocated and, if found, increases its reference counter by one. When a FORGET request arrives for an inode, Stackfs decreases inode's reference count and deallocates the inode when the count drops to zero.

File create and open. During file creation, Stackfs adds a new inode to the hash table after the corresponding file was successfully created in the underlying file system. While processing OPEN requests, Stackfs saves the file descriptor of the underlying file in the file handle. The file descriptor is then used during read and write operations and deallocated when the file is closed.

3.2 Performance Statistics and Traces

The existing FUSE instrumentation was insufficient for in-depth FUSE performance analysis. We therefore instrumented FUSE to export important runtime statistics. Specifically, we were interested in recording the duration of time that FUSE spends in various stages of request processing, both in kernel and user space.

We introduced a two-dimensional array where a row index (0–42) represents the request type and the column index (0–31) represents the time. Every cell in the array stores the number of requests of a corresponding type that were processed within the 2^{N+1}–2^{N+2} nanoseconds where N is the column index. The time dimension therefore covers the interval of up to 8 seconds which is enough in typical FUSE setups. (This technique efficiently records a log_2 latency histogram [18].) We then added four such arrays to FUSE: the first three arrays are in the kernel, capturing the time spent by the request inside the background, pending, and processing queues. For the processing queue, the captured time also includes the time spent by requests in user space. The fourth array is in user space and tracks the time the daemon needs to process a request. The total memory size of all four arrays is only 48KiB and only few instructions are necessary to update values in the array.

FUSE includes a special `fusectl` file system to allow users to control several aspects of FUSE's behavior. This file system is usually mounted at `/sys/fs/fuse/connections/` and creates a directory for every mounted FUSE instance. Every directory contains control files to abort a connection, check the total number of requests being processed, and adjust the upper limit and the threshold on the number of background requests (see Section 2.2). We added 3 new files to these directories to export statistics from the in-kernel arrays. To export user-level array we added `SIGUSR1` signal handler to the daemon. When triggered, the handler prints the array to a log file specified during the daemon's start. The statistics captured have no measurable overhead on FUSE's performance and are the primary source of information about FUSE's performance.

Tracing. To understand FUSE's behavior in more detail we sometimes needed more information and had to resort to tracing. FUSE's library already performs tracing when the daemon runs in debug mode but there is no tracing support for FUSE's kernel module. We used Linux's static tracepoint mechanism [10] to add over 30 tracepoints mainly to monitor the formation of requests during the complex writeback logic, reads, and some metadata operations. Tracing helped us learn how fast queues grow during our experiments, how much data is put into a single request, and why.

Both FUSE's statistics and tracing can be used by any existing and future FUSE-based file systems. The instrumentation is completely transparent and requires no changes to file-system-specific code.

4 Methodology

FUSE has evolved significantly over the years and added several useful optimizations: writeback cache, zero-copy via splicing, and multi-threading. In our personal experience, some in the storage community tend to pre-judge FUSE's performance—assuming it is poor—mainly due to not having information about the improvements FUSE made over the years. We therefore designed our methodology to evaluate and demonstrate how FUSE's performance advanced from its basic configurations to ones that include all of the latest optimizations. We now detail our methodology, starting from the description of FUSE configurations, proceed to the list of workloads, and finally present our testbed.

FUSE configurations. To demonstrate the evolution of FUSE's performance, we picked two configurations on opposite ends of the spectrum: the *basic* configuration (called *StackfsBase*) with no major FUSE optimizations and the *optimized* configuration (called *StackfsOpt*) that enables all FUSE improvements available as of this writing. Compared to StackfsBase, the StackfsOpt configuration adds the following features: (1) writeback cache is turned on; (2) maximum size of a single FUSE request is increased from 4KiB to

128KiB (max_write parameter); (3) user daemon runs in the multi-threaded mode; (4) splicing is activated for all operations (splice_read, splice_write, and splice_move parameters). We left all other parameters at their default values in both configurations.

Workloads. To stress different modes of FUSE operation and conduct an thorough performance characterization, we selected a broad set of workloads: micro and macro, metadata- and data-intensive, and also experimented with a wide range of I/O sizes and parallelism levels. Table 2 describes all workloads that we employed. To simplify the identification of workloads in the text we use the following mnemonics: rnd stands for random, seq for sequential, rd for reads, wr for writes, cr for creates, and del for deletes. The presence of *N*th and *M*f substrings in a workload name means that the workload contains *N* threads and *M* files, respectively. We fixed the amount of work (e.g., the number of reads in rd workloads) rather than the amount of time in every experiment. We find it easier to analyze performance in experiments with a fixed amount of work. We picked a sufficient amount of work so that the performance stabilized. Resulting runtimes varied between 8 and 20 minutes across the experiments. Because SSDs are orders of magnitude faster than HDDs, for some workloads we selected a larger amount of work for our SSD-based experiments. We used Filebench [12, 37] to generate all workloads.

Experimental setup. FUSE performance depends heavily on the speed of the underlying storage: faster devices expose FUSE's own overheads. We therefore experimented with two common storage devices of different speed: an HDD (Seagate Savvio 15K.2, 15KRPM, 146GB) and an SSD (Intel X25-M SSD, 200GB). Both devices were installed in three identical Dell PowerEdge R710 machines with 4-core Intel Xeon E5530 2.40GHz CPU each. The amount of RAM available to the OS was set to 4GB to accelerate cache warmup in our experiments. The machines ran CentOS 7 with Linux kernel upgraded to v4.1.13 and FUSE library commit #386b1b.

We used Ext4 [11] as the underlying file system because it is common, stable, and has a well documented design which facilitates performance analysis. Before every experiment we reformatted the storage devices with Ext4 and remounted the file systems. To lower the variability in our experiments we disabled Ext4's lazy inode initialization [5]. In either case, standard deviations in our experiments were less than 2% for all workloads except for three: seq-rd-1th-1f (6%), files-rd-32th (7%), and mail-server (7%).

5 Evaluation

For many, FUSE is just a practical tool to build real products or prototypes, but not a research focus. To present our results more effectively, we split the evaluation in two. Section 5.1 overviews our extensive evaluation results—most useful information for many practitioners. Detailed performance analysis follows in Section 5.2.

5.1 Performance Overview

To evaluate FUSE's performance degradation, we first measured the throughput (in ops/sec) achieved by native Ext4 and then measured the same for Stackfs deployed over Ext4. As detailed in Section 4 we used two configurations of Stackfs: a basic (*StackfsBase*) and optimized (*StackfsOpt*) one. From here on, we use Stackfs to refer to both of these configurations. We then calculated the relative performance degradation (or improvement) of Stackfs vs. Ext4 for each workload. Table 3 shows absolute throughputs for Ext4 and relative performance for two Stackfs configurations for both HDD and SSD.

For better clarity we categorized the results by Stackfs's performance difference into four classes: (1) The *Green* class (marked with $^{+}$) indicates that the performance either degraded by less than 5% or actually improved; (2) The *Yellow* class (*) includes results with the performance degradation in the 5–25% range; (3) The *Orange* class ($^{\#}$) indicates that the performance degradation is between 25–50%; And finally, (4) the *Red* class ($^{!}$) is for when performance decreased by more than 50%. Although the ranges for acceptable performance degradation depend on the specific deployment and the value of other benefits provided by FUSE, our classification gives a broad overview of FUSE's performance. Below we list our main observations that characterize the results. We start from the general trends and move to more specific results towards the end of the list.

Observation 1. The relative difference varied across workloads, devices, and FUSE configurations from –83.1% for files-cr-1th [row #37] to +6.2% for web-server [row #45].

Observation 2. For many workloads, FUSE's optimizations improve performance significantly. E.g., for the web-server workload, StackfsOpt improves performance by 6.2% while StackfsBase degrades it by more than 50% [row #45].

Observation 3. Although optimizations increase the performance of some workloads, they can degrade the performance of others. E.g., StackfsOpt decreases performance by 35% more than StackfsBase for the files-rd-1th workload on SSD [row #39].

Observation 4. In the best performing configuration of Stackfs (among StackfsOpt and StackfsBase) only two file-create workloads (out of a total 45 workloads) fell into the red class: files-cr-1th [row #37] and files-cr-32th [row #38].

Workload Name	Description	Results
`seq-rd-Nth-1f`	*N* threads (1, 32) sequentially read from a single preallocated 60GB file.	[rows #1–8]
`seq-rd-32th-32f`	32 threads sequentially read 32 preallocated 2GB files. Each thread reads its own file.	[rows #9–12]
`rnd-rd-Nth-1f`	*N* threads (1, 32) randomly read from a single preallocated 60GB file.	[rows #13–20]
`seq-wr-1th-1f`	Single thread creates and sequentially writes a new 60GB file.	[rows #21–24]
`seq-wr-32th-32f`	32 threads sequentially write 32 new 2GB files. Each thread writes its own file.	[rows #25–28]
`rnd-wr-Nth-1f`	*N* threads (1, 32) randomly write to a single preallocated 60GB file.	[rows #29–36]
`files-cr-Nth`	*N* threads (1, 32) create 4 million 4KB files over many directories.	[rows #37–38]
`files-rd-Nth`	*N* threads (1, 32) read from 1 million preallocated 4KB files over many directories.	[rows #39–40]
`files-del-Nth`	*N* threads (1, 32) delete 4 million of preallocated 4KB files over many directories.	[rows #41–42]
`file-server`	File-server workload emulated by Filebench. Scaled up to 200,000 files.	[row #43]
`mail-server`	Mail-server workload emulated by Filebench. Scaled up to 1.5 million files.	[row #44]
`web-server`	Web-server workload emulated by Filebench. Scaled up to 1.25 million files.	[row #45]

Table 2: Description of workloads and their corresponding result rows. For data-intensive workloads, we experimented with 4KB, 32KB, 128KB, and 1MB I/O sizes. We picked dataset sizes so that both cached and non-cached data are exercised. The Results column correlates these descriptions with results in Table 3.

Observation 5. Stackfs's performance depends significantly on the underlying device. E.g., for sequential read workloads [rows #1–12], Stackfs shows no performance degradation for SSD and a 26–42% degradation for HDD. The situation is reversed, e.g., when a `mail-server` [row #44] workload is used.

Observation 6. At least in one Stackfs configuration, all write workloads (sequential and random) [rows #21–36] are within the *Green* class for both HDD and SSD.

Observation 7. The performance of sequential read [rows #1–12] are well within the *Green* class for both HDD and SSD; however, for the `seq-rd-32th-32f` workload [rows #5–8] on HDD, they are in *Orange* class. Random read workload results [rows #13–20] span all four classes. Furthermore, the performance grows as I/O sizes increase for both HDD and SSD.

Observation 8. In general, Stackfs performs visibly worse for metadata-intensive and macro workloads [rows #37–45] than for data-intensive workloads [rows #1–36]. The performance is especially low for SSDs.

Observation 9. The relative CPU utilization of Stackfs (not shown in the Table) is higher than that of Ext4 and varies in the range of +0.13% to +31.2%; similarly, CPU cycles per operation increased by 1.2× to 10× times between Ext4 and Stackfs (in both configurations). This behavior is seen in both HDD and SSD.

Observation 10. CPU cycles per operation are higher for StackfsOpt than for StackfsBase for the majority of workloads. But for the workloads `seq-wr-32th-32f` [rows #25–28] and `rnd-wr-1th-1f` [rows #30–32], StackfsOpt consumes fewer CPU cycles per operation.

5.2 Analysis

We analyzed FUSE performance results and present main findings here, following the order in Table 3.

5.2.1 Read Workloads

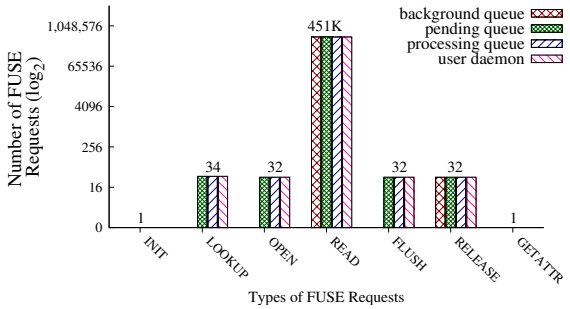

Figure 3: Different types and number of requests generated by StackfsBase on SSD during the `seq-rd-32th-32f` workload, from left to right, in their order of generation.

Figure 3 demonstrates the types of requests that were generated with the `seq-rd-32th-32f` workload. We use `seq-rd-32th-32f` as a reference for the figure because this workload has more requests per operation type compared to other workloads. Bars are ordered from left to right by the appearance of requests in the experiment. The same request types, but in different quantities, were generated by the other read-intensive workloads [rows #1–20]. For the single threaded read workloads, only one request per LOOKUP, OPEN, FLUSH, and RELEASE type was generated. The number of READ requests depended on the I/O size and the amount of data read; INIT request is produced at mount time so its count remained the same across all workloads; and finally GETATTR is invoked before unmount for the root directory and was the same for all the workloads.

Figure 3 also shows the breakdown of requests by queues. By default, READ, RELEASE, and INIT are asynchronous; they are added to the background queue first. All other requests are synchronous and are added to pending queue directly. In read workloads, only READ requests are generated in large numbers. Thus, we discuss in detail only READ requests for these workloads.

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			EXT4 (ops/sec)	StackfsBase (%Diff)	StackfsOpt (%Diff)	EXT4 (ops/sec)	StackfsBase (%Diff)	StackfsOpt (%Diff)
1	seq-rd-1th-1f	4	38382	- 2.45^{+}	+ 1.7^{+}	30694	- 0.5^{+}	- 0.9^{+}
2		32	4805	- 0.2^{+}	- 2.2^{+}	3811	+ 0.8^{+}	+ 0.3^{+}
3		128	1199	- 0.86^{+}	- 2.1^{+}	950	+ 0.4^{+}	+ 1.7^{+}
4		1024	150	- 0.9^{+}	- 2.2^{+}	119	+ 0.2^{+}	- 0.3^{+}
5	seq-rd-32th-1f	4	1228400	- 2.4^{+}	- 3.0^{+}	973450	+ 0.02^{+}	+ 2.1^{+}
6		32	153480	- 2.4^{+}	- 4.1^{+}	121410	+ 0.7^{+}	+ 2.2^{+}
7		128	38443	- 2.6^{+}	- 4.4^{+}	30338	+ 1.5^{+}	+ 1.97^{+}
8		1024	4805	- 2.5^{+}	- 4.0^{+}	3814.50	- 0.1^{+}	- 0.4^{+}
9	seq-rd-32th-32f	4	11141	- 36.9$^{\#}$	- 26.9$^{\#}$	32855	- 0.1^{+}	- 0.16^{+}
10		32	1491	- 41.5$^{\#}$	- 30.3$^{\#}$	4202	- 0.1^{+}	- 1.8^{+}
11		128	371	- 41.3$^{\#}$	- 29.8$^{\#}$	1051	- 0.1^{+}	- 0.2^{+}
12		1024	46	- 41.0$^{\#}$	- 28.3$^{\#}$	131	- 0.03^{+}	- 2.1^{+}
13	rnd-rd-1th-1f	4	243	- 9.96*	- 9.95*	4712	- 32.1$^{\#}$	- 39.8$^{\#}$
14		32	232	- 7.4*	- 7.5*	2032	- 18.8*	- 25.2$^{\#}$
15		128	191	- 7.4*	- 5.5*	852	- 14.7*	- 12.4*
16		1024	88	- 9.0*	-3.1^{+}	114	- 15.3*	-1.5^{+}
17	rnd-rd-32th-1f	4	572	- 60.4$^{!}$	-23.2*	24998	- 82.5$^{!}$	-27.6$^{\#}$
18		32	504	- 56.2$^{!}$	-17.2*	4273	- 55.7$^{!}$	-1.9^{+}
19		128	278	- 34.4$^{\#}$	-11.4*	1123	- 29.1$^{\#}$	-2.6^{+}
20		1024	41	- 37.0$^{\#}$	-15.0*	126	- 12.2*	-1.9^{+}
21	seq-wr-1th-1f	4	36919	-26.2$^{\#}$	- 0.1^{+}	32959	- 9.0*	+ 0.1^{+}
22		32	4615	- 17.8*	- 0.16^{+}	4119	- 2.5^{+}	+ 0.12^{+}
23		128	1153	- 16.6*	- 0.15^{+}	1030	- 2.1^{+}	+ 0.1^{+}
24		1024	144	- 17.7*	-0.31^{+}	129	- 2.3^{+}	- 0.08^{+}
25	seq-wr-32th-32f	4	34370	- 2.5^{+}	+ 0.1^{+}	32921	+ 0.05^{+}	+ 0.2^{+}
26		32	4296	- 2.7^{+}	+ 0.0^{+}	4115	+ 0.1^{+}	+ 0.1^{+}
27		128	1075	- 2.6^{+}	- 0.02^{+}	1029	- 0.04^{+}	+ 0.2^{+}
28		1024	134	- 2.4^{+}	- 0.18^{+}	129	- 0.1^{+}	+ 0.2^{+}
29	rnd-wr-1th-1f	4	1074	- 0.7^{+}	- 1.3^{+}	16066	+ 0.9^{+}	- 27.0$^{\#}$
30		32	708	- 0.1^{+}	- 1.3^{+}	4102	- 2.2^{+}	- 13.0*
31		128	359	- 0.1^{+}	- 1.3^{+}	1045	- 1.7^{+}	- 0.7^{+}
32		1024	79	- 0.01^{+}	- 0.8^{+}	129	- 0.02^{+}	- 0.3^{+}
33	rnd-wr-32th-1f	4	1073	- 0.9^{+}	- 1.8^{+}	16213	- 0.7^{+}	- 26.6$^{\#}$
34		32	705	+ 0.1^{+}	- 0.7^{+}	4103	- 2.2^{+}	- 13.0*
35		128	358	+ 0.3^{+}	- 1.1^{+}	1031	- 0.1^{+}	+ 0.03^{+}
36		1024	79	+ 0.1^{+}	- 0.3^{+}	128	+ 0.9^{+}	- 0.3^{+}
37	files-cr-1th	4	30211	- 57$^{!}$	- 81.0$^{!}$	35361	- 62.2$^{!}$	- 83.3$^{!}$
38	files-cr-32th	4	36590	- 50.2$^{!}$	- 54.9$^{!}$	46688	- 57.6$^{!}$	- 62.6$^{!}$
39	files-rd-1th	4	645	+ 0.0^{+}	- 10.6*	8055	- 25.0*	- 60.3$^{!}$
40	files-rd-32th	4	1263	- 50.5$^{!}$	-4.5^{+}	25341	- 74.1$^{!}$	-33.0$^{\#}$
41	files-del-1th	-	1105	- 4.0^{+}	- 10.2*	7391	- 31.6$^{\#}$	- 60.7$^{!}$
42	files-del-32th	-	1109	- 2.8^{+}	- 6.9*	8563	- 42.9$^{\#}$	- 52.6$^{!}$
43	file-server	-	1705	- 26.3$^{\#}$	-1.4^{+}	5201	- 41.2$^{\#}$	-1.5^{+}
44	mail-server	-	1547	- 45.0$^{\#}$	-4.6^{+}	11806	- 70.5$^{!}$	-32.5$^{\#}$
45	web-server	-	1704	- 51.8$^{!}$	+6.2^{+}	19437	- 72.9$^{!}$	-17.3*

Table 3: List of workloads and corresponding performance results. Green *class (marked with $^{+}$) indicates that the performance either degraded by less than 5% or actually improved;* Yellow *class (*) includes results with the performance degradation in the 5–25% range;* Orange *class ($^{\#}$) indicates that the performance degradation is between 25–50%; And finally, the* Red *class ($^{!}$) is for when performance decreased by more than 50%.*

Sequential Read using 1 thread on 1 file. The total number of READ requests that StackfsBase generated during the whole experiment for different I/O sizes for HDD and SSD remained approximately the same and equal to 491K. Our analysis revealed that this happens because of FUSE's default 128KB-size readahead which effectively levels FUSE request sizes no matter what is the user application I/O size. Thanks to readahead, sequential read performance of StackfsBase and StackfsOpt was as good as Ext4 for both HDD and SSD.

Sequential Read using 32 threads on 32 files. Due to readahead, the total number of READ requests generated here was also approximately same for different I/O sizes. At any given time, 32 threads are requesting data and continuously add requests to queues. StackfsBase and StackfsOpt show significantly larger performance degradation on HDD compared to SSD. For StackfsBase, the user daemon is single threaded and the device is slower, so requests do not move quickly through the queues. On the faster SSD, however, even though the user daemon is single threaded, requests move faster in the queues. Hence performance of StackfsBase is as close to that of Ext4. With StackfsOpt, the user daemon is multi-threaded and can fill the HDD's queue faster so performance improved for HDD compared to SSD.

Investigating further, we found that for HDD and StackfsOpt, FUSE's daemon was bound by the *max_background* value (default is 12): at most, only 12 user deamons (threads) were spawned. We increased that limit to 100 and reran the experiments: now StackfsOpt was within 2% of Ext4's performance.

Sequential Read using 32 threads on 1 file. This workload exhibits similar performance trends to `seq-rd-1th-1f`. However, because all 32 user threads read from the same file, they benefit from the shared page cache. As a result, instead of 32× more FUSE requests, we saw only up to a 37% increase in number of requests. This modest increase is because, in the beginning of the experiment, every thread tries to read the data separately; but after a certain point in time, only a single thread's requests are propagated to the user daemon while all other threads' requests are available in the page cache. Also, having 32 user threads running left less CPU time available for FUSE's threads to execute, thus causing a slight (up to 4.4%) decrease in performance compared to Ext4.

Random Read using 1 thread on 1 file. Unlike the case of small sequential reads, small random reads did not benefit from FUSE's readahead. Thus, every application read call was forwarded to the user daemon which resulted in an overhead of up to 10% for HDD and 40% for SSD. The absolute Ext4 throughput is about 20× higher for SSD than for HDD which explains the higher penalty on FUSE's relative performance on SSD.

The smaller the I/O size is, the more READ requests are generated and the higher FUSE's overhead tended to be. This is seen for StackfsOpt where performance for HDD gradually grows from –10.0% for 4KB to –3% for 1MB I/O sizes. A similar situation is seen for SSD. Thanks to splice, StackfsOpt performs better than StackfsBase for large I/O sizes. For 1MB I/O size, the improvement is 6% on HDD and 14% on SSD. Interestingly, 4KB I/O sizes have the highest overhead because FUSE splices requests only if they are larger than 4KB.

Random Read using 32 threads on 1 file. Similar to the previous experiment (single thread random read), readahead does not help smaller I/O sizes here: every user read call is sent to the user daemon and causes high performance degradation: up to –83% for StackfsBase and –28% for StackfsOpt. The overhead caused by StackfsBase is high in these experiments (up to –60% for HDD and –83% for SSD), for both HDD and SSD, and especially for smaller I/O sizes. This is because when 32 user threads submit a READ request, 31 of those threads need to wait while the single-threaded user daemon processes one request at a time. StackfsOpt reduced performance degradation compared to StackfsBase, but not as much for 4KB I/Os because splice is not used for request that are smaller or equal to 4KB.

5.2.2 Write Workloads

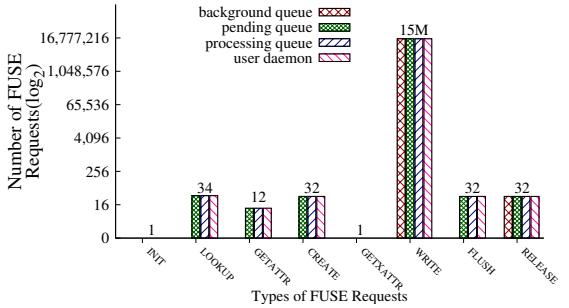

Figure 4: Different types of requests that were generated by StackfsBase on SSD for the `seq-wr-32th-32f` workload, from left to right in their order of generation.

We now discuss the behavior of StackfsBase and StackfsOpt in all write workloads listed in Table 3 [rows #21–36]. Figure 4 shows the different types of requests that got generated during all write workloads, from left to right in their order of generation (`seq-wr-32th-32f` is used as a reference). In case of `rnd-wr` workloads, CREATE requests are replaced by OPEN requests, as random writes operate on pre-allocated files. For all the `seq-wr` workloads, due to the creation of files, a GETATTR request was generated to check permissions of the single directory where the files were created. Linux VFS caches attributes and therefore there were fewer than 32 GETATTRs. For single-threaded

Stages of `write()` call processing	Time (μs)	Time (%)
Processing by VFS before passing execution to FUSE kernel code	1.4	2.4
FUSE request allocation and initialization	3.4	6.0
Waiting in queues and copying to user space	10.7	18.9
Processing by Stackfs daemon, includes Ext4 execution	24.6	43.4
Processing reply by FUSE kernel code	13.3	23.5
Processing by VFS after FUSE kernel code	3.3	5.8
Total	56.7	100.0

Table 4: Average latencies of a single write request generated by StackfsBase during seq-wr-4KB-1th-1f workload across multiple profile points on HDD.

workloads, five operations generated only one request: LOOKUP, OPEN, CREATE, FLUSH, and RELEASE; however, the number of WRITE requests was orders of magnitude higher and depended on the amount of data written. Therefore, we consider only WRITE requests when we discuss each workload in detail.

Usually the Linux VFS generates GETXATTR before every write operation. But in our case StackfsBase and StackfsOpt did not support extended attributes and the kernel cached this knowledge after FUSE returned ENOSUPPORT for the first GETXATTR.

Sequential Write using 1 thread on 1 file. The total number of WRITE requests that StackfsBase generated during this experiment was 15.7M for all I/O sizes. This is because in StackfsBase each user write call is split into several 4KB-size FUSE requests which are sent to the user daemon. As a result StackfsBase degraded performance ranged from –26% to –9%. Compared to StackfsBase, StackfsOpt generated significantly fewer FUSE requests: between 500K and 563K depending on the I/O size. The reason is the writeback cache that allows FUSE's kernel part to pack several dirty pages (up to 128KB in total) into a single WRITE request. Approximately $\frac{1}{32}$ of requests were generated in StackfsOpt compared to StackfsBase. This suggests indeed that each WRITE request transferred about 128KB of data (or 32× more than 4KB).

Table 4 shows the breakdown of time spent (latencies) by a single write request across various stages, during the `seq-wr-4KB-1th-1f` workload on HDD. Taking only major latencies, the write request spends 19% of its time in request creation and waiting in the kernel queues; 43% of its time in user space, which includes time taken by the underlying Ext4 to serve the write; and then 23% of time during copy of the response from user space to kernel. The relative CPU utilization caused by StackfsBase and StackfsOpt in `seq-wr-4KB-1th-1f` on HDD is 6.8% and 11.1% more than native Ext4, respectively; CPU cycles per operation were the same for StackfsBase and StackfsOpt—4× that of native Ext4.

Sequential Write using 32 threads on 32 files. Performance trends are similar to `seq-wr-1th-1f` but even the unoptimized StackfsBase performed much better (up to –2.7% and –0.1% degradation for HDD and SSD, respectively). This is because without the writeback cache, 32 user threads put more requests into FUSE's queues (compared to 1 thread) and therefore kept the user daemon constantly busy.

Random Write using 1 thread on 1 file. Performance degradation caused by StackfsBase and StackfsOpt was low on HDD for all I/O sizes (max –1.3%) because the random write performance of Ext4 on HDD is low—between 79 and 1,074 Filebench ops/sec, depending on the I/O size (compare to over 16,000 ops/sec for SSD). The performance bottleneck, therefore, was in the HDD I/O time and FUSE overhead was invisible.

Interestingly, on SSD, StackfsOpt performance degradation was high (–27% for 4KB I/O) and more than the StackfsBase for 4KB and 32KB I/O sizes. The reason for this is that currently FUSE's writeback cache batches only *sequential* writes into a single WRITE. Therefore, in the case of *random* writes there is no reduction in the number of WRITE requests compared to StackfsBase. These numerous requests are processed asynchronously (i.e., added to the background queue). And because of FUSE's congestion threshold on the background queue the application that is writing the data becomes throttled.

For I/O size of 32KB, StackfsOpt can pack the entire 32KB into a single WRITE request. Compared to StackfsBase, this reduces the number of WRITE requests by 8× and results in 15% better performance.

Random Write using 32 threads on 1 file. This workload performs similarly to `rnd-wr-1th-1f` and the same analysis applies.

5.2.3 Metadata Workloads

We now discuss the behavior of Stackfs in all metadata micro-workloads as listed in Table 3 [rows #37–42].

File creates. Different types of requests that got generated during the `files-cr-Nth` runs are GETATTR, LOOKUP, CREATE, WRITE, FLUSH, RELEASE, and FORGET. The total number of each request type generated was exactly 4 million. Many GETATTR requests were generated due to Filebench calling a `fstat` on the file to check whether it exists or not before creating it. `Files-cr-Nth` workloads demonstrated the worst performance among all workloads for both StackfsBase and StackfsOpt and for both HDD and SSD. The reason is twofold. First, for every single file create, five operations happened serially: GETATTR, LOOKUP, CREATE, WRITE, and FLUSH; and as there were many files accessed, they all could not be cached, so we saw many FORGET requests to remove cached items—which added further overhead. Second, file creates are fairly fast in

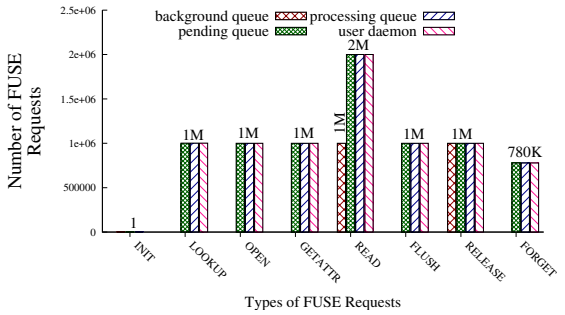

Figure 5: Different types of requests that were generated by StackfsBase on SSD for the `files-rd-1th` workload, from left to right in their order of generation.

Ext4 (30–46 thousand creates/sec) because small newly created inodes can be effectively cached in RAM. Thus, overheads caused by the FUSE's user-kernel communications explain the performance degradation.

File Reads. Figure 5 shows different types of requests that got generated during the `files-rd-1th` workload. This workload is metadata-intensive because it contains many small files (one million 4KB files) that are repeatedly opened and closed. Figure 5 shows that half of the READ requests went to the background queue and the rest directly to the pending queue. The reason is that when reading a whole file, and the application requests reads beyond the EOF, FUSE generates a synchronous READ request which goes to the pending queue (not the background queue). Reads past the EOF also generate a GETATTR request to confirm the file's size.

The performance degradation for `files-rd-1th` in StackfsBase on HDD is negligible; on SSD, however, the relative degradation is high (–25%) because the SSD is 12.5× faster than HDD (see Ext4 absolute throughput in Table 3). Interestingly, StackfsOpt's performance degradation is more than that of StackfsBase (by 10% and 35% for HDD and SSD, respectively). The reason is that in StackfsOpt, *different* FUSE threads process requests for the *same* file, which requires additional synchronization and context switches. Conversely, but as expected, for `files-rd-32th` workload, StackfsOpt performed 40–45% better than StackfsBase because multiple threads are needed to effectively process parallel READ requests.

File Deletes. The different types of operations that got generated during the `files-del-1th` workloads are LOOKUP, UNLINK, FORGET (exactly 4 million each). Every UNLINK request is followed by FORGET. Therefore, for every incoming delete request that the application (Filebench) submits, StackfsBase and StackfsOpt generates three requests (LOOKUP, UNLINK, and FORGET) in series, which depend on each other.

Deletes translate to small random writes at the block layer and therefore Ext4 benefited from using an SSD (7–8× higher throughput than the HDD). This negatively impacted Stackfs in terms of relative numbers: its performance degradation was 25–50% higher on SSD than on HDD. In all cases StackfsOpt's performance degradation is more than StackfsBase's because neither splice nor the writeback cache helped `files-del-Nth` workloads and only added additional overhead for managing extra threads.

5.2.4 Macro Server Workloads

We now discuss the behavior of Stackfs for macro-workloads [rows #43–45].

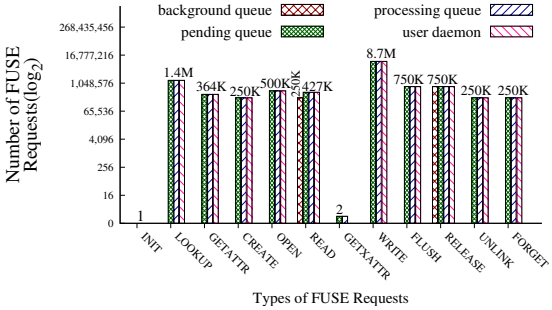

Figure 6: Different types of requests that were generated by StackfsBase on SSD for the `file-server` workload.

File Server. Figure 6 shows different types of operations that got generated during the `file-server` workload. Macro workloads are expected to have a more diverse request profile than micro workloads, and `file-server` confirms this: many different requests got generated, with WRITEs being the majority.

The performance improved by 25–40% (depending on storage device) with StackfsOpt compared to StackfsBase, and got close to Ext4's native performance for three reasons: (1) with a writeback cache and 128KB requests, the number of WRITEs decreased by a factor of 17× for both HDD and SSD, (2) with splice, READ and WRITE requests took advantage of zero copy, and (3) the user daemon is multi-threaded, as the workload is.

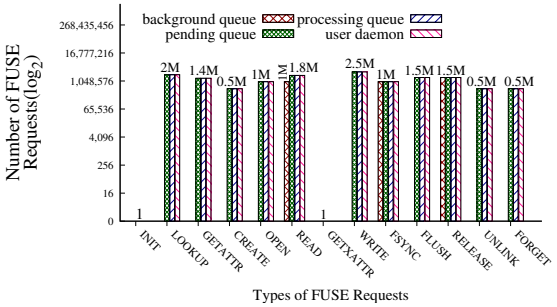

Figure 7: Different types of requests that were generated by StackfsBase on SSD for the `mail-server` workload.

Mail Server. Figure 7 shows different types of operations that got generated during the `mail-server` workload. As with the `file-server` workload, many

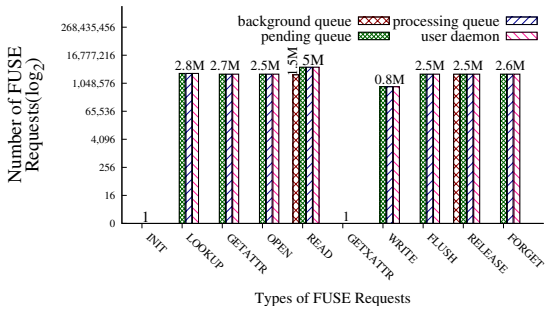

Figure 8: Different types of requests that were generated by StackfsBase on SSD for the `web-server` *workload.*

different requests got generated, with WRITEs being the majority. Performance trends are also similar between these two workloads. However, in the SSD setup, even the optimized StackfsOpt still did not perform close to Ext4 in this `mail-server` workload, compared to `file-server`. The reason is twofold. First, compared to file server, mail server has almost double the metadata operations, which increases FUSE overhead. Second, I/O sizes are smaller in mail-server which improves the underlying Ext4 SSD performance and therefore shifts the bottleneck to FUSE.

Web Server. Figure 8 shows different types of requests generated during the `web-server` workload. This workload is highly read-intensive as expected from a Web-server that services static Web-pages. The performance degradation caused by StackfsBase falls into the *Red* class in both HDD and SSD. The major bottleneck was due to the FUSE daemon being single-threaded, while the workload itself contained 100 user threads. Performance improved with StackfsOpt significantly on both HDD and SSD, mainly thanks to using multiple threads. In fact, StackfsOpt performance on HDD is even 6% *higher* than of native Ext4. We believe this minor improvement is caused by the Linux VFS treating Stackfs and Ext4 as two independent file systems and allowing them together to cache more data compared to when Ext4 is used alone, without Stackfs. This does not help SSD setup as much due to the high speed of SSD.

6 Related Work

Many researchers used FUSE to implement file systems [3, 9, 15, 40] but little attention was given to understanding FUSE's underlying design and performance. To the best of our knowledge, only two papers studied some aspects of FUSE. First, Rajgarhia and Gehani evaluated FUSE performance with Java bindings [27]. Compared to this work, they focused on evaluating Java library wrappers, used only three workloads, and ran experiments with FUSE v2.8.0-pre1 (released in 2008). The version they used did not support zero-copying via splice, writeback caching, and other important features. The authors also presented only limited information about FUSE design at the time.

Second, in a position paper, Tarasov et al. characterized FUSE performance for a variety of workloads but did not analyze the results [36]. Furthermore, they evaluated only default FUSE configuration and discussed only FUSE's high-level architecture. In this paper we evaluated and analyzed several FUSE configurations in detail, and described FUSE's low-level architecture.

Several researchers designed and implemented useful extensions to FUSE. Re-FUSE automatically restarts FUSE file systems that crash [33]. To improve FUSE performance, Narayan et al. proposed to marry in-kernel stackable file systems [44] with FUSE [23]. Shun et al. modified FUSE's kernel module to allow applications to access storage devices directly [17]. These improvements were in research prototypes and were never included in the mainline.

7 Conclusion

User-space file systems are popular for prototyping new ideas and developing complex production file systems that are difficult to maintain in kernel. Although many researchers and companies rely on user-space file systems, little attention was given to understanding the performance implications of moving file systems to user space. In this paper we first presented the detailed design of FUSE, the most popular user-space file system framework. We then conducted a broad performance characterization of FUSE and we present an in-depth analysis of FUSE performance patterns. We found that for many workloads, an optimized FUSE can perform within 5% of native Ext4. However, some workloads are unfriendly to FUSE and even if optimized, FUSE degrades their performance by up to 83%. Also, in terms of the CPU utilization, the relative increase seen is 31%.

All of our code and Filebench workloads files are available from *http://filesystems.org/fuse/*.

Future work. There is a large room for improvement in FUSE performance. We plan to add support for compound FUSE requests and investigate the possibility of shared memory between kernel and user spaces for faster communications.

Acknowledgments

We thank the anonymous FAST reviewers and our shepherd Tudor Marian for their valuable comments. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, and CNS-1622832; and ONR award 12055763.

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*, pages 93–112, Atlanta, GA, June 1986. USENIX Association.

[2] The Apache Foundation. Hadoop, January 2010. *http://hadoop.apache.org*.

[3] 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/*.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, Network Working Group, June 1995.

[5] Z. Cao, V. Tarasov, H. Raman, D. Hildebrand, and E. Zadok. On the performance variation in modern storage stacks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February/March 2017. USENIX Association. to appear.

[6] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Seattle, WA, December 1994.

[7] Michael Condict, Don Bolinger, Dave Mitchell, and Eamonn McManus. Microkernel Modularity with Integrated Kernel Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI 1994)*, Monterey, CA, November 1994.

[8] Jonathan Corbet. In defense of per-bdi writeback, September 2009. *http://lwn.net/Articles/354851/*.

[9] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 19–28, Boston, MA, June 2004. USENIX Association.

[10] Mathieu Desnoyers. Using the Linux kernel tracepoints, 2016. *https://www.kernel.org/doc/Documentation/trace/tracepoints.txt*.

[11] Ext4 Documentation. *https://www.kernel.org/doc/Documentation/filesystems/ext4.txt*.

[12] Filebench, 2016. *https://github.com/filebench/filebench/wiki*.

[13] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003. ACM SIGOPS.

[14] Hermann Hartig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schonberg. The performance of Microkernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP '97)*, Saint Malo, France, October 1997. ACM.

[15] V. Henson, A. Ven, A. Gud, and Z. Brown. Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep 2006)*, Seattle, WA, November 2006. ACM SIGOPS.

[16] GNU Hurd. *www.gnu.org/software/hurd/hurd.html*.

[17] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. Optimizing local file accesses for FUSE-based distributed storage. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pages 760–765. IEEE, 2012.

[18] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[19] Lessfs, January 2012. *www.lessfs.com*.

[20] Linus Torvalds doesn't understand user-space filesystems. *http://redhatstorage.redhat.com/2011/06/28/linus-torvalds-doesnt-understand-user-space-storage/*.

[21] David Mazieres. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.

[22] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[23] Sumit Narayan, Rohit K Mehta, and John A Chandy. User space storage system stack modules with file level control. In *Proceedings of the 12th Annual Linux Symposium in Ottawa*, pages 189–196, 2010.

[24] Nimble's Hybrid Storage Architecture. *http://info.nimblestorage.com/rs/nimblestorage/images/nimblestorage_technology_overview.pdf*.

[25] NTFS-3G. *www.tuxera.com*.

[26] David Pease, Arnon Amir, Lucas Villa Real, Brian Biskeborn, Michael Richmond, and Atsushi Abe. The linear tape file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[27] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *25th Symposium On Applied Computing*. ACM, March 2010.

[28] Nikolaus Rath. List of fuse based file systems (git page), 2011. *https://github.com/libfuse/libfuse/wiki/Filesystems*.

[29] Glusterfs. *http://www.gluster.org/*.

[30] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[31] Opendedup, January 2012. *www.opendedup.org*.

[32] 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*, Anaheim, CA, June 1990. IEEE.

[33] Swaminathan Sundararaman, Laxman Visampalli, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Refuse to crash with Re-FUSE. In *Proceedings of the sixth conference on Computer systems*, pages 77–90. ACM, 2011.

[34] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

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

[36] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok. Terra incognita: On the practicality of user-space file systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*, Santa Clara, CA, July 2015.

[37] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[38] L. Torvalds. *splice()*. Kernel Trap, 2007. *http://kerneltrap.org/node/6505*.

[39] Linux Torwalds. Re: [patch 0/7] overlay filesystem: request for inclusion. *https://lkml.org/lkml/2011/6/9/462*.

[40] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *Proceedings of the FAST Conference*, 2010.

[41] Sage Weil. Linus vs fuse. *http://ceph.com/dev-notes/linus-vs-fuse/*.

[42] Assar Westerlund and Johan Danielsson. Arla-a free AFS client. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, New Orleans, LA, June 1998. USENIX Association.

[43] Felix Wiemann. List of fuse based file systems (wiki page), 2006. *https://en.wikipedia.org/wiki/Filesystem_in_Userspace#Example_uses*.

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

[45] ZFS for Linux, January 2016. *www.zfs-fuse.net*.

[46] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, USA, 2008.