# Stony Brook University

# To FUSE or not to FUSE? Analysis and Performance Characterization of the FUSE User-Space File System Framework

A Thesis Presented

by

**Bharath Kumar Reddy Vangoor**

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

**Master of Science**

in

**Computer Science**

Stony Brook University

**Technical Report FSL-16-02**

**December 2016**

**Stony Brook University**

The Graduate School

**Bharath Kumar Reddy Vangoor**

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

**Dr. Erez Zadok, Thesis Advisor**
Professor, Computer Science

**Dr. Mike Ferdman, Thesis Committee Chair**
Assistant Professor, Computer Science

**Dr. Vasily Tarasov**
IBM Research – Almaden

This thesis is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Thesis

**To FUSE or not to FUSE? Analysis and Performance Characterization of the FUSE User-Space File System Framework**

by

**Bharath Kumar Reddy Vangoor**

**Master of Science**

in

**Computer Science**

Stony Brook University

**December 2016**

Traditionally, file systems were implemented as part of operating systems kernels, which provide a limited set of tools and facilities to a programmer. As complexity of file systems grew, many new file systems began being developed in user space. 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 thesis we analyze the design and implementation of the most widely known user-space file system framework, FUSE, for Linux; we characterize its performance for a wide range of workloads. We present FUSE performance with various mount and configuration options, using 45 different workloads that were generated using Filebench on two different hardware configurations. 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 non-existent or as high as minus 83%, even when optimized. Our thesis is that user-space file systems can indeed be used in production (non "toy") settings, but their applicability depends on the expected workloads.

To my Mother Rajitha;
Father Bal Reddy;
Sister Priyanka, and Family;
Mamayya Sridhar Reddy, and Family;
Grandparents Nanamma, Thathaya, and Ammamma; and
Herta and Wolfgang.

# Contents

**List of Figures** vii

**List of Tables** vii

**Acknowledgments** ix

**1 Introduction** 1

**2 FUSE Design** 4
- 2.1 High-Level Architecture . . . 4
- 2.2 User-Kernel Protocol . . . 6
- 2.3 Library and API Levels . . . 8
- 2.4 Queues . . . 10
- 2.5 Splicing and FUSE Buffers . . . 11
- 2.6 Notifications . . . 12
- 2.7 Multithreading . . . 13
- 2.8 Linux Write-back Cache and FUSE . . . 13
  - 2.8.1 Linux Page Cache . . . 13
  - 2.8.2 Page Cache Parameters . . . 14
  - 2.8.3 Write Back Cache Code Flow . . . 16
  - 2.8.4 FUSE write-back and max write . . . 17
- 2.9 Miscellaneous Options . . . 18

**3 Implementations** 20
- 3.1 Stackfs . . . 20
- 3.2 Performance Statistics and Traces . . . 22
  - 3.2.1 Tracing . . . 22
- 3.3 Development Effort . . . 23

**4 Methodology** 25
- 4.1 Configurations . . . 25
- 4.2 Workloads . . . 26
- 4.3 Experimental Setup . . . 27

**5 Evaluation** 28
- 5.1 Performance Overview . . . 28
- 5.2 Analysis . . . 32
  - Read Workloads . . . 32

Write Workloads . . . . . 35
Metadata Workloads . . . . . 36
Macro Server Workloads . . . . . 38

**6 Related Work** **41**

**7 Conclusions** **42**

**Bibliography** **43**

# List of Figures


2.1 FUSE high-level architecture. . . . . 5
2.2 Interaction of FUSE library levels. “My File System” is an illustrative user-space file system implemented with the high-level API. . . . . 9
2.3 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. . . . . 10

3.1 Locations of tracepoints throughout the flow of FUSE requests. . . . . 23

5.1 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. . . . 32
5.2 Total number of READ requests that were generated by StackfsBase on HDD and SSD for the `seq-rd-1th-1f` workload. . . . . 33
5.3 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. . . . . 35
5.4 Different types of requests that were generated by StackfsBase on SSD for the `files-cr-1th` workload, from left to right in their order of generation. . . . . 36
5.5 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. . . . . 37
5.6 Different types of requests that were generated by StackfsBase on SSD for the `files-del-1th` workload, from left to right in their order of generation. . . . . 38
5.7 Different types of requests that were generated by StackfsBase on SSD for the `file-server` workload. . . . . 39
5.8 Different types of requests that were generated by StackfsBase on SSD for the `mail-server` workload. . . . . 39
5.9 Different types of requests that were generated by StackfsBase on SSD for the `web-server` workload. . . . . 40

# List of Tables


2.1 FUSE request types grouped by semantics. The number in parenthesis is the size of the corresponding group. Request types that we discuss in the text are typeset in bold. 6
2.2 FUSE notification types, in the order of their opcodes. . . . . 12
2.3 Parameter names, default values (if applicable), and their shortcut names used throughout this thesis. . . . . 15
2.4 Library Options . . . . . 18
2.5 Low-Level Options . . . . . 19

3.1 Development effort in project. WML is Filebench's Workload Model Language. . . . 24

4.1 Description of workloads. For data-intensive workloads, we experimented with 4KB, 32KB, 128KB, and 1MB I/O sizes. We picked data-set sizes so that both cached and non-cached data are exercised. . . . . 26

5.1 List of workloads and corresponding performance results. Stackfs1 refers to StackfsBase and Stackf2 refers to StackfsOpt. . . . . 29
5.2 List of workloads and corresponding CPU utilization in secs. FUSE1 refers to StackfsBase and FUSE2 refers to StackfsOpt. . . . . 30
5.3 List of workloads and corresponding performance results. Stackfs1 refers to StackfsBase, Stackfs2 refers to StackfsOpt, and Stackfs3 referes to StackfsBase with increased background_queue limit. . . . . 34

# Acknowledgments

This work could not have been possible without the efforts and sacrifices of a lot of people.

My adviser, Dr. Erez Zadok has been extremely supportive throughout my time at the File Systems and Storage Lab (FSL). He was very involved with this work and gave invaluable suggestions.

Dr. Vasily Tarasov, Research Staff Member at IBM Research–Almaden, helped me with the project in several ways. He was instrumental at offering immense help to this project with coding, debugging, reviewing, benchmarking, and project management. He did all of this on top of his full time job and personal responsibilities. He is such an inspiration, and I thank him profusely for everything.

Arun, a masters student at FSL, also worked hard to get work done in relatively less time. We thank Arun for his contributions.

I also acknowledge DongJu, Garima, Jason, Ming, Zhen and all the other students at FSL. We spent quality time with each other and discovered a lot.

Let me also appreciate Prof. Anshul Gandhi (who taught Performance Analysis of Systems, and with whom I did my Independent Study), Prof. Rob Johnson (Algorithms), and Prof. Annie Liu (Asynchronous Systems). The courses with them improved my knowledge of systems. Oracle's patching division in Hyderabad, who I worked for, laid the foundation for my understanding of systems and especially ZFS. I extend a special recognition to Balaji, Shyam, and the team.

I would also like to thank my undergraduate (NIT-Calicut) professors, especially Prof. Subhasree who encouraged and supported me during my graduate school applications. I should also mention about my friends from my undergraduate days – Narayan Rao, Naik, Uday, Raghavender, Sudeep, Gautham, Deepak, Shashank, Burri, Praveen, Gupta, and many more who are still in contact with me and constantly appreciating my progress.

Finally, I am grateful to the valuable feedback by committee members – Dr. Mike Ferdman, Committee Chair; Dr. Vasily Tarasov; and Dr. Erez Zadok, Thesis Adviser.

This work was made possible in part thanks to EMC, NetApp, and IBM support; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, and CNS-1622832; and ONR award 12055763.

# Chapter 1

# Introduction

The file system is one of the oldest and perhaps most common interfaces for applications to access their data. Although in the years of micro-kernel-based Operating Systems (OS), some file systems were implemented in user space [1, 28], the status quo was always to implement file systems as part of the monolithic OS kernels [12,36,49]. Kernel-space implementations avoid potentially high overheads of passing requests between the kernel and user-space daemons—communications that are inherent to the user-space and micro-kernel implementations [14, 26].

However, slowly although perhaps not overly noticeable, user-space file systems have crawled back into today's storage systems. In recent years, user-space file systems rose in popularity and following are some of the proofs (indicators/signs) to support the discussion:

1. A number of user-space stackable file systems that add specialized functionalities on top of basic file systems gained popularity (e.g., deduplication and compression file systems [31, 46]).

2. In academic research and advanced development, user-space file system frameworks like FUSE became a de facto standard for experimenting with new approaches to file system design [8, 16, 27, 56].

3. Several existing kernel-level file systems were ported to user space (e.g., ZFS [60], NTFS [40]). Some tried to bring parts of file system to user-space as specialized solutions [47, 58] or port to Windows [2].

4. Perhaps most important, an increasing number of companies rely on user-space implementations for their storage products: IBM'S GPFS [45] and LTFS [41], Nimble Storage's CASL [39], Apache's HDFS [4], Google File System [25], RedHat's GlusterFS [44], Data Domain's deduplication file system [61], and more. Some implement file systems in user space to store data online in clouds using services like Google Drive, Amazon S3 [43], and DropBox [37].

Although user-space file systems did not displace kernel-level file systems entirely, and it would be incorrect and premature to assume this, user-space file systems undoubtedly occupy a growing niche.

Customers constantly demand new features in storage solutions (snapshotting, deduplication, encryption, automatic tiering, replication, etc.). Vendors reply to this demand by releasing new products with a vastly increased complexity. For example, when expressed as the Lines of Code (LoC), recent Btrfs versions contain over 85,000 LoC—at least 5× more than the already classic Ext3 file system (about 15,000 LoC, including journaling code). For modern distributed file systems, especially ones supporting multiple platforms, the LoC count reaches several millions (e.g., IBM's GPFS [45]). With the continuous advent of Software Defined Storage (SDS) [13] paradigms, the amount of code and storage-software complexity is only expected to grow.

Increased file systems complexity is a major factor in user-space file systems' growing popularity. The user space is a much friendlier environment to develop, port, and maintain the code. NetBSD's *rump kernel* is the first implementation of the "anykernel" concept where drivers can be run in user space on top of a light-weight rump kernel; these drivers included file systems as well [21]. A number of frameworks for writing user-space file systems appeared [3, 5, 6, 10, 20, 34, 35]. We now discuss some of the reasons for the rise in the popularity of user-space file systems in recent times:

1. **Developement ease.** Developers' toolboxes now include numerous user-space tools for tracing, debugging, and profiling user programs. Accidental user-level bugs do not crash the whole system and do not require a lengthy reboot but rather generate a useful core-memory dump while the program can be easily restarted for further investigation. A myriad of useful libraries are readily available in user-space. Developers are not limited to only few system-oriented programming languages (e.g., C) but can easily use several higher-level languages, each best suited to its goal.

2. **Portability.** For file systems that need to run on multiple platforms, it is much easier to develop portable code in user space than in the kernel. This can be recognized in case of distributed file systems, whose clients often run on multiple OSes [58]. E.g., if file systems were developed in user space initially, then UNIX file systems could have been readily accessed under windows.

3. **Libraries.** An abundance of libraries are available in user space where it is easier to try new and more efficient algorithms for implementing a file system, to improve performance. For example, predictive prefetching can use AI algorithms to adapt to a specific user's workload; and cache management can use classification libraries to implement better eviction policies.

Simply put, there are many more developers readily available to code in user space than in the kernel and the entry bar is much lower for newcomers.

Of course, everything that can be done in user space can be achieved in the kernel. But to keep the development scalable with the complexity of file systems, many companies resort to user-space implementations. The more visible this comeback of user-space file systems becomes, the more heated are the debates between proponents and opponents of user-space file systems [32, 55, 57]. While some consider user-space file systems just a toy, others develop real production systems with them. The article in the official Red Hat Storage blog titled "Linus Torvalds doesn't understand user-space filesystems" is indicative of these debates [32].

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 has several well defined metrics and can be systematically evaluated. Oddly, little has been published on the performance of user-space file system frameworks.

In this thesis, we use the most common user-space file system framework, *FUSE*, on Linux, and characterize the performance degradation it causes and its resource utilization. We start with a detailed explanation of FUSE's design and implementation for four reasons:

1. FUSE's architecture is somewhat complex;

2. Little information on FUSE's internals is available publicly;

3. FUSE's source code can be overwhelming to analyze, with complex asynchrony between user-level and kernel-level components;

4. As user-space file systems and FUSE grow in popularity, a detailed analysis of their implementation becomes of high value to at least two groups: (a) engineers in industry, as they design and build new systems; and (b) researchers, because they use FUSE for prototyping new solutions.

   and finally

5. Understanding FUSE design is crucial for the analysis presented in this thesis.

We developed a simple pass-through stackable file system using FUSE, called *StackFS*, which we layer on top of Ext4. We evaluated its performance compared to the native Ext4 using a wide variety of micro- and macro-workloads running on different hardware. In addition, we measured the increase in CPU utilization caused by FUSE. Our findings indicate that depending on the workload and hardware used, FUSE can perform as good as native Ext4 file system; but in the worst cases, FUSE can perform $3\times$ slower than the underlying Ext4 file system. In terms of raw CPU cycles, FUSE's CPU utilization increases by up to $13\times$, but relative utilization increases by 31%.

Next, we designed and implemented a rich instrumentation system for FUSE that allows us to conduct in-depth performance analysis. The statistics extracted are applicable to any FUSE-based systems. We released our code publicly and can be found at following locations, all accessible from *http://www.filesystems.org/fuse*:

1. Fuse Kernel Instrumentation:
   - Repo: `https://github.com/sbu-fsl/fuse-kernel-instrumentation.git`
   - Branch: `master`
2. Fuse Library Instrumentation:
   - Repo: `https://github.com/sbu-fsl/fuse-library-instrumentation.git`
   - Branch: `master`
3. Workloads, Results, and Stackable File system implementation:
   - Repo: `https://github.com/sbu-fsl/fuse-stackfs.git`
   - Branch: `master`

Additional information about code can be found at the aforementioned link, `http://filesystems.org/fuse/`. We then used this instrumentation to identify bottlenecks in FUSE and explain why it performs well for some workloads while struggles for others. For example, we demonstrate that currently FUSE cannot prefetch or compound small random reads and therefore performs poorly for workloads with many such small operations.

The rest of this thesis is organized as follows. Chapter 2 discusses the FUSE architecture and implementation details. Chapter 3 describes our stackable file system's implementation and useful instrumentation that we added to FUSE's kernel and user library. Chapter 4 introduces our experimental methodology. The bulk of our evaluation and analysis is in Chapter 5. Then we discuss the related work in the Chapter 6. We conclude and describe future work in Chapter 7.

# Chapter 2

# FUSE Design

FUSE—Filesystem in Userspace—is the most widely used user-space file system framework [50]. Initial implementation was originally developed for Linux-based OSes but over time it was ported to many other OSes [33, 60]. According to the most modest estimates, at least 100 FUSE-based file systems are readily available on the Web [51]. Although other, specialized implementations of user-space file systems exist [45, 47, 58], we selected FUSE for this study because of its high popularity. We believe that our findings here will not only impact a large family of already existing FUSE file systems, but can also be applied to other user-space file system frameworks.

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 [42]. For our evaluation it was essential to understand not only FUSE's high-level design but also some intricate 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 (65 files changed, 2,680 insertions, 3,693 deletions) 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.

Next, we detail FUSE's design, starting with a high level architecture in Section 2.1. We describe how FUSE user space daemon and kernel driver communicate in Section 2.2. Different API levels available in FUSE user library are discussed in Section 2.3. Different types of queues which are part of FUSE kernel driver are described in Section 2.4. Section 2.5 describes the FUSE capability of zero-copy using splice and special buffers that are maintained internally. Section 2.6 discusses FUSE's notification subsystem. Section 2.7 describes how FUSE's library supports multi-threading and processes requests in parallel. In Section 2.8 we discuss important parameters associated with FUSE's write-back in-kernel cache ,and how these values affect FUSE. Finally, in Section 2.9 we discuss various command-line options that FUSE currently supports.

## 2.1 High-Level Architecture

FUSE consists of a kernel part and a user-level daemon. The kernel part is implemented as a Linux kernel module `fuse.ko` which, when loaded, registers three file system types with the Virtual File System (VFS) (all visible in `/proc/filesystems`): 1) *fuse*, 2) *fuseblk*, and 3) *fusectl*. Both *fuse* and *fuseblk* are proxy types for various specific FUSE file systems that are implemented by different

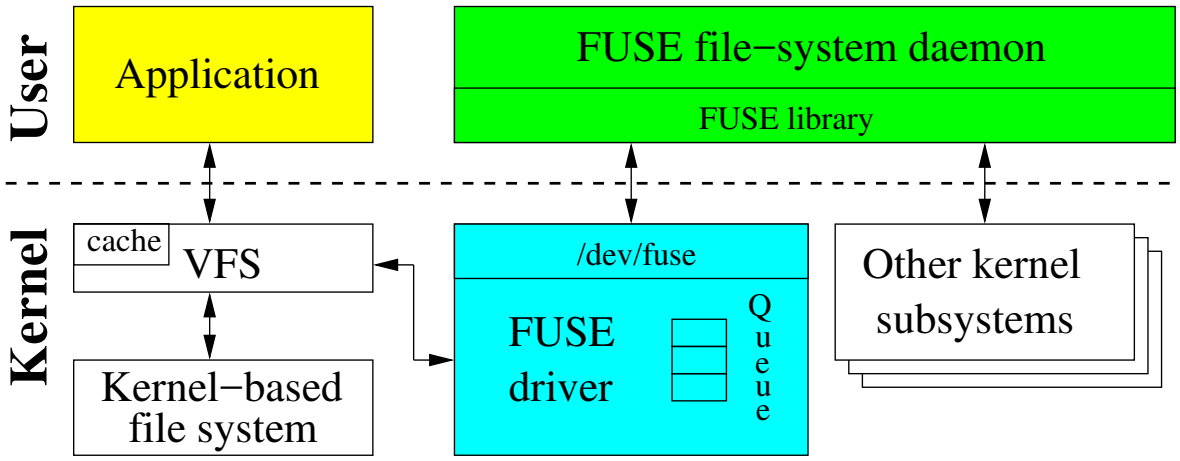


Figure 2.1: FUSE high-level architecture.

user-level daemons. File systems of *fuse* type do not require underlying block devices and are usually stackable, in-memory, or network file systems. The *fuseblk* type, on the other hand, is for user-space file systems that are deployed on top of block devices, much like traditional local file systems. The *Fuse* and *fuseblk* types are similar in implementation; for a single-device file systems, however, the *fuseblk* provides following features:

1. Locking the block device on mount and unlocking on release;
2. Sharing the file system for multiple mounts;
3. Allowing swap files to bypass the file system in accessing the underlying device;

In addition to these features, *fuseblk* also provides the ability to synchronously unmount the file system: i.e, when the file system is last to be unmounted (no lazy unmounts or bind mounts remain), then the unmount call will wait until the file system acknowledges this (e.g., flushes buffers). We discuss this behaviour later in Section 2.2. We refer to both *fuse* and *fuseblk* as FUSE from here on. Finally, the *fusectl* file system provides users with the means to control and monitor any FUSE file system behavior (e.g., for setting thresholds and counting the number of pending requests).

The *fuse* and *fuseblk* file system types are different from traditional file systems (e.g., ext4 or XFS) in that they present whole families of file systems. To differentiate between different *mounted* FUSE file systems, the `/proc/mounts` file represents every specific FUSE file system as `[fuse|fuseblk].`*<NAME>* (instead of just `[fuse|fuseblk]`). The *<NAME>* is a string identifier specified by the FUSE file-system developer (e.g.,"dedup" if a file system deduplicates data).

In addition to registering three file systems, 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, the daemon reads FUSE requests from `/dev/fuse`, processes them, and then writes replies back to `/dev/fuse`.

Figure 2.1 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 (file system) 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; and in case of a network or in-memory file system, the FUSE daemon might still need to re-enter the kernel to obtain certain system services (e.g., create a socket or get the time of day). 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 which submitted the request.

Some file system operations invoked by an application can complete without communicating with the user-level FUSE daemon. For example, reads from a previously read file, whose pages are cached in the kernel page cache, do not need to be forwarded to the FUSE driver. Caching is not limited to data: meta-data information (e.g., for `stat(2)`) about cached inodes or dentries (cached in Linux's dcache) can be fully processed in kernel space without involving a FUSE daemon (depending on the mount options).

We chose a stackable user-space file system instead of a block, network, or in-memory one for our study because the majority of existing FUSE-based file systems are stackable (i.e., deployed on top of other, often in-kernel file systems). Moreover, we wanted to add as little overhead as possible so as to isolate the overhead of FUSE's kernel and library.

| **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 2.1: FUSE request types grouped by semantics. The number in parenthesis is the size of the corresponding group. Request types that we discuss in the text are typeset in bold.

## 2.2 User-Kernel Protocol

When FUSE's kernel driver communicates to the user-space daemon, it forms a *FUSE request* structure. The header files `include/uapi/linux/fuse.h` (for use in kernel code) and `include/fuse_kernel.h` (for use in FUSE's user-level library) are identical; they define the requests headers which are common to the user library and the kernel, thereby providing kernel-user interoperability. In particular, `struct fuse_in_header` and `fuse_out_header` define common headers for all input requests (to the daemon) and output replies (from the daemon). Similarly, they define operation-specific headers: `struct fuse_read_in/fuse_read_out` (for read operations), `struct fuse_fsync_in/out` (for fsync operations), etc. Requests have different types (stored in the `opcode` field of the `fuse_in_header`) depending on the operation they convey.

FUSE's user-kernel protocol also provides a mechanism to store some information per opened file/directory. The ability to store or not store information makes the protocol statefull and stateless, respectively. For each operation (in the low-level API) dealing with files/directories, there is a `struct fuse_file_info` passed as an argument. This structure is maintained by the FUSE li-

brary to track the information about files in the FUSE file system (user space). This structure contains an unsigned 64-bit integer field `fh` (file handle) which can be used to store information about opened files (stateful). For example, while replying to requests like OPEN and CREATE, FUSE's daemon can store a file descriptor in this field. The advantage of this is that the `fh` is then passed by the kernel to the user daemon for all the operations associated with the opened file. If this is not set in the first place, then FUSE's daemon has to open and close the file for every file operation (statelessness). This holds true even for requests like OPENDIR and MKDIR where a directory pointer associated with an opened directory can be type-casted and stored in the `fh`.

Table 2.1 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 focus next on those less intuitive request types (marked in bold in Table 2.1).

The INIT request is produced by the kernel when a file system is mounted. At this point the user space and kernel negotiate the following three items:

1. The protocol version they will operate on (v7.23 at the time of this writing);
2. The set of mutually supported capabilities (e.g., READDIRPLUS or FLOCK support), where `FUSE_CAP_*` contains all possible capabilities of user/kernel; and
3. Various parameter settings (e.g., FUSE read-ahead size, time granularity for attributes) which are passed as mount options.

Conversely, the DESTROY request is sent by the kernel during the file system's unmounting process. When getting a DESTROY request, 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. Currently, a DESTROY request is sent synchronously only in case of *fuseblk* but not in case of *fuse* file systems types (as mentioned in Section 2.1). The reason for this behavior is because *fuseblk* is mounted with a privileged user credentials (access to block device); and in this case, unmount needs to handle flushing all the buffers during unmount. Unmounting in the case of a *fuse* file system, however, need not wait on the unprivileged FUSE daemon [23].

The INTERRUPT request is emitted by the kernel if any requests that were previously passed to the daemon are no longer needed (e.g., when a user process blocked on a READ request is terminated). INTERRUPT requests take precedence over other requests, so the user-space file system will receive queued INTERRUPT requests before any other requests. The user-space file system may ignore the INTERRUPT requests entirely, or may honor them by sending a reply to the original request, with the error set to `EINTR`. 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 back to the kernel. Every request also contains a *node ID*—an unsigned 64-bit integer identifying the inode both in kernel and user spaces (sometimes referred to as an inode ID). The path-to-inode translation is performed by the LOOKUP request. FUSE's root inode number is always 1. Every time an existing inode is looked up (or a new one is created), the kernel keeps the inode in the inode and directory entry cache (dcache). Several requests exist to facilitate management of caches in the FUSE daemon, to mirror their respective dcache states in the kernel. When removing an inode from the dcache, the kernel passes the FORGET request to the user-space daemon. FUSE's inode reference count (in user space file system) grows with every reply to LOOKUP, CREATE, etc. requests. FORGET request passes an `nlookups` parameter which informs the user-space file system (daemon) about how many lookups to forget. At this point the user space file system (daemon) might decide to deallocate any corresponding data structures (once their reference count goes to 0). BATCH_FORGET sends batched requests to forget multiple inodes.

An OPEN request is generated, not surprisingly, when a user application opens a file. When replying to this request (as already discussed in this section), a FUSE daemon has a chance to optionally assign a 64-bit *file handle* to the opened file. This file handle is then passed by the kernel to the daemon along with every request associated with the opened file. The user-space daemon can use the handle to store per-opened-file information. For example, a stackable FUSE file system can store the descriptor of the file opened in the underlying file system as part of FUSE's file handle. It is not necessary to set a file handle; if not set, the protocol will be stateless. 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. One RELEASE request is generated for every OPEN request (when closed), and there might be multiple FLUSHs per OPEN because of forks, dups, etc.

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 meta-data information for each entry. This allows the kernel to pre-fill its inode cache, similar to NFSv3's READDIRPLUS procedure [9].

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 the kernel to grant or deny access to a file based on its standard Unix attributes (ownership and permission bits). In this case (*default_permissions*) no ACCESS requests are generated.

RENAME2 just adds a flags field compared to RENAME; flags that are currently supported are as follows:

1. `RENAME_NOREPLACE`: this flag indicates that if the target of the rename exists, then rename should fail with `EEXIST` instead of replacing the target (as expected by POSIX).

2. `RENAME_EXCHANGE`: this flag indicates that both source and target must exist but may be of different types (e.g., one could be a non-empty directory and the other a symbolic link). If either of them doesn't exists, then rename should fail with `ENOENT` instead of replacing the target. If no flags are passed, then the kernel falls back to using a RENAME request.

Both FSYNC and FSYNCDIR are used to synchronize data and meta-data on files and directories, respectively. These requests also carry a additional flag with them, `datasync`, which allows developers to differentiate when to sync data and meta-data. That is, when the `datasync` parameter is non-zero, then only user data is flushed, not the meta-data.

FUSE supports file locking using following request types:

1. GETLK checks to see if there is already a lock on the file, but doesn't set one.

2. SETLKW obtains the requested lock. If the lock cannot be obtained (e.g., someone else has it locked already), then wait (block) until the lock has cleared and then grab the lock for yourself.

3. SETLK is almost identical to SETLKW. The only difference is that it will not wait if it cannot obtain a lock. Instead, it returns immediately with an error.

## 2.3 Library and API Levels

Conceptually, the FUSE library consists of two levels, as seen in Figure 2.2. The lower level takes care of the following:

1. Receiving and parsing requests from the kernel,

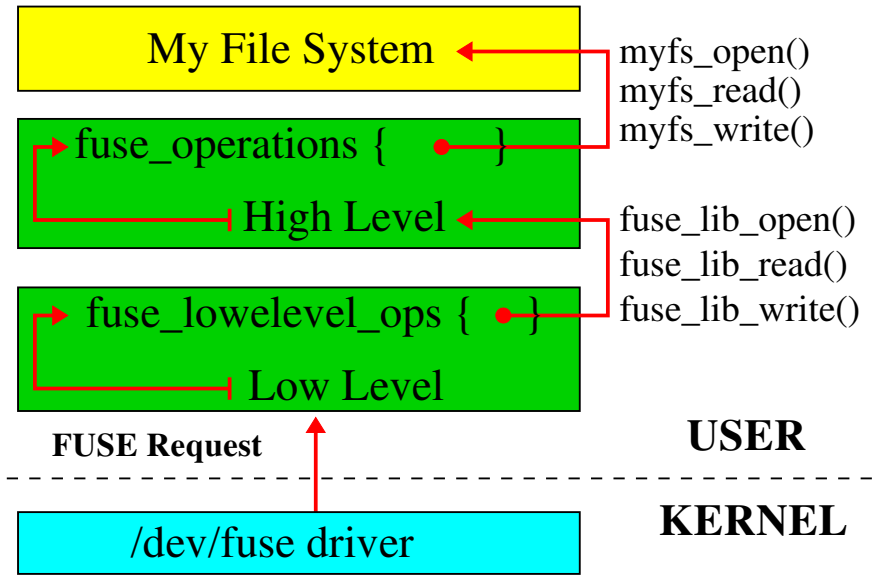


Figure 2.2: Interaction of FUSE library levels. "My File System" is an illustrative user-space file system implemented with the high-level API.

2. Sending properly formatted replies,
3. Facilitating file system configuration and mounting, and
4. Hiding potential version differences between kernel and user spaces.

This part of the library exposes to developers the so-called *low-level FUSE API*.

The *High-level FUSE API* builds on top of the low-level API and allows file system developers to skip the implementation of the *path-to-inode* mapping. Therefore, the high-level API omits the complex concept of lookup operations; inodes do not exist at this high-level API, simplifying code development for many FUSE file-system developers. Instead of using inodes, all high-level API methods operate directly on file paths. As a result, FORGET inode methods are not needed in the high-level API. 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()`. The high-level API never communicates with the kernel directly, only through the low-level API. The low-level API implemented within the library has function names like `fuse_lib_*()` (e.g., `fuse_lib_read`, etc.). These functions internally call high-level API functions, depending upon the functionality.

File system developers must decide which API to use, by balancing flexibility vs. development ease. For simpler file systems that do not need to handle lookups, a high-level API is a good fit. In this case, developers need to implement 42 methods in the `fuse_operations` structure. These methods roughly correspond to traditional POSIX file system operations, e.g., open, read, write, mkdir; and almost all of them take a file name as one of the arguments. If a developer decides to use the low-level API, then 42 methods in the `fuse_lowlevel_ops` need to be implemented. Many methods in both structures are optional.

The methods in both APIs are similar and we highlight important differences of low-level API compared to the high-level API below. First, for higher flexibility, low-level methods always take a FUSE request as an argument. Second, method definitions have even closer match to Linux VFS methods because they often operate on inodes (or rather inode numbers). For example, the lookup

method is present in low-level API and takes (in addition to complete FUSE request) the parent inode number and the name of the file to be looked up. Third, paired with a lookup method is an additional `forget` method that is called when a kernel removes an inode from the inode/dentry cache.

Figure 2.2 schematically describes the interaction of FUSE's library levels. When the low level API receives a request from the kernel, it parses the request and calls an appropriate method in `fuse_lowlevel_ops`. The methods in this structure are either implemented by the user file system itself (if the low-level API was selected to develop the user file system) or directly by the high level of FUSE's library. In the latter case, the high-level part of the library calls an appropriate method in `fuse_operations` structure, which is implemented by the file system that is developed with FUSE's high-level API.

## 2.4 Queues

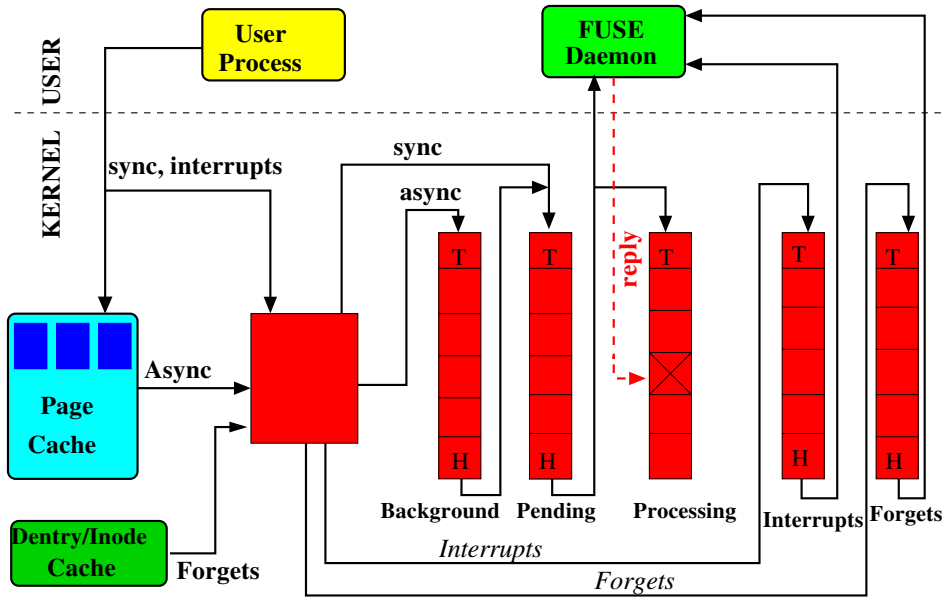


Figure 2.3: The organization of FUSE queues marked with their Head and Tail. The processing queue does not have a tail because the daemon replies in an arbitrary order.

In Section 2.1 we briefly mentioned that FUSE's kernel part maintains a request queue for processing by the user space. More specifically, FUSE maintains five queues as seen in Figure 2.3: (1) *interrupts* (2) *forgets* (3) *pending*, (4) *processing*, and (5) *background*. At any point in time a request belongs to only one queue. FUSE puts INTERRUPT requests in the interrupts queue, FORGET requests in the forgets queue, and synchronous requests (e.g., metadata) in the pending queue. FUSE uses a separate queue for INTERRUPT requests for assigning priority (to interrupt requests); and similarly, FUSE uses a separate queue for FORGET requests to differentiate them from non-forget requests. FORGET requests are sent when the inode is evicted, and these requests are would queue up together with regular file system requests, if a separate forgets queue did not exist. If many FORGET requests are processed, then no other file system operation can proceed. This behavior was seen when a FUSE file system with 32 million inodes on a machine with lots of memory can become unresponsive for up to 30 minutes when all those inodes are evicted from the icache [24]. Therefore, FUSE maintains a separate queue for FORGET requests and a fair policy is implemented to process them, as explained

below.

When a file-system (user space) daemon reads from `/dev/fuse`, then the requests are transferred to the user daemon as follows:

1. First priority is given to the requests (if any) in the interrupts queue; i.e., the oldest INTERRUPT request is transferred to the user space before any other requests.

2. FORGET and non-FORGET requests are selected fairly: for each 8 non-FORGET requests, 16 FORGET requests are transferred. This ensures that FORGET requests do not pile up, yet other file systems requests are also allowed to proceed while the queued forgets are processed.

The oldest request in the pending queue is transferred to the user space and simultaneously moved to the processing queue. INTERRUPT and FORGET requests do not have a reply (from user daemon); therefore, as soon as user daemon reads these requests, they are terminated. Thus, requests in the processing queue are the ones that are currently processed by the daemon. If the pending queue is empty then the FUSE daemon is blocked on a read call. When the daemon replies to the request (by writing the reply to `/dev/fuse`), the corresponding request is removed from the processing queue, which concludes the life of the request. At the same time, blocked user processes (e.g., the ones waiting for READ to complete) are notified that they can proceed.

The background queue is for staging asynchronous requests. In a default setup (no arguments to FUSE on mount), only read requests go to the background queue because by default reads are asynchronous as they read more than the process requested due to read_ahead; writes go to the background queue but only if the writeback cache is enabled. In addition, FUSE puts INIT requests and RELEASE requests into the background queue. When the writeback cache is enabled, writes from the user process are first accumulated in the page cache and later `bdflush` threads wake up to flush dirty pages [15]. 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 and also to limit the number of user daemon threads invoked in case of multi-threaded option (which is discussed in Section 2.7).

The queues' lengths are not explicitly limited: 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 submit requests to this file system.

## 2.5 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 [54]. Splicing (represented by the `splice()` system call family) allows the user space to transfer data between two in-kernel memory buffers without copying the data to user space. This is useful 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. The regular memory region identified by a pointer in the user daemon's address space, or

2. The kernel-space memory pointed by a file descriptor of a pipe where the data resides.

If a user-space file system implements the `write_buf()` method (in the low-level API), then FUSE first splices the data from `/dev/fuse` to a Linux pipe and then passes the data directly to this method as a buffer containing a file descriptor of the pipe. FUSE splices only WRITE requests and only the ones that contain more than a single page of data. Similar logic applies to the replies to READ requests if the `read_buf()` method is implemented. However, the `read_buf` method is only present in the high-level API; for the low-level API, the file-system developer has to differentiate between splice and non-splice flows inside the `read` method itself.

If the library is compiled with splice support, the kernel supports it, and appropriate command-line parameters are set, then `splice()` is always called for every request (including the request's header). However, the header of every single request needs to be examined, for example to identify the request's type and size. This examination is not possible if the FUSE buffer has only the file descriptor of a pipe where the data resides. So, for every request the header is then read from the pipe using regular `read()` calls (i.e., small, at most 80 bytes, memory copying is always perfomed). FUSE then splices the requested data if its size is larger than a single page (excluding the header): therefore only big writes are spliced. For reads, replies larger than two pages are spliced.

## 2.6 Notifications

| **Notification Types** | **Description** |
|---|---|
| POLL | Notifies that an event happened on an opened file for which an application has polled on. |
| INVAL_INODE | Invalidates cache for a specific inode. |
| INVAL_ENTRY | Invalidates parent directory attributes and the dentry matching $\langle$*parent-directory*, *file-name*$\rangle$ pair. |
| STORE | Store data in the page cache. The STORE and RETRIEVE notification types can be used by FUSE file systems to synchronize their caches with the kernel's. |
| RETRIEVE | Retrieve data from the page cache. The STORE and RETRIEVE notification types can be used by FUSE file systems to synchronize their caches with the kernel's. |
| DELETE | Notify to invalidate parent attributes and delete the dentry matching parent/-name if the dentry's inode number matches the child (otherwise it will invalidate the matching dentry). |

Table 2.2: FUSE notification types, in the order of their opcodes.

So far we did not discuss any mechanisms for the FUSE library to communicate to its kernel counterpart, except by replying to a kernel request. But in certain situations, the daemon needs to pass some information to the kernel without receiving any previous requests from the kernel. For example, if a user application polls events on an opened file using the `poll()` system call, the FUSE daemon needs to notify the kernel when an event happens that wakes up the waiting process. For this and similar situations, FUSE introduces the concept of *notifications*. As with replies, to pass a notification to the kernel, the daemon writes the notification to `/dev/fuse`.

Table 2.2 describes the six notification types that the FUSE daemon can send to the kernel driver. Usually, FUSE's kernel driver does not send any replies to notifications. The only exception is the RETRIEVE request, for which the kernel replies using a special NOTIFY_REPLY request.

## 2.7 Multithreading

Initial FUSE implementations supported only a single-threaded daemon but as parallelism got more dominant, both in user applications and hardware, the necessity of multithreading became evident. When executed in multithreaded mode, FUSE at first starts only one thread. However, 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 is 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. The implicit limit arises due to two factors. First, by default, only 12 asynchronous requests (`max_background` parameter) can be in the pending queue at one time. Second, the number of synchronous requests in the pending queue is constrained by the number of user processes that submit requests. In addition, for every INTERRUPT and FORGET requests, a new thread is invoked. Therefore, the total number of FUSE daemon threads is at most $(12 + number\ of\ processes\ with\ outstanding\ I/O + number\ of\ interrupts + no\ of\ forgets)$. But in a typical system where there is no interrupts support and not many forgets are generated, the total number of FUSE daemon threads are at most $(12 + number\ of\ processes\ with\ outstanding\ I/O)$.

## 2.8 Linux Write-back Cache and FUSE

Section 2.8.1 describes Linux's Page Cache internals and Section 2.8.2 describes its parameters. Section 2.8.3 describes FUSE's write-back cache code flow. Section 2.8.4 describes FUSE's write-back mode.

### 2.8.1 Linux Page Cache

The Linux kernel implements a cache called page cache [53]. The main advantage of this cache is to minimize disk I/O by storing data in physical memory (RAM) that would otherwise require a disk access. The page cache consists of physical pages in RAM. These pages originate from reads and writes of file system files, block device files, and memory-mapped files. Therefore, the page cache contains recently accessed file data. During an I/O operation such as a read, the kernel first checks whether the data is present in the page cache. If so, the kernel then quickly returns the requested page from memory rather than read the data from the slower disk. If the data is read from disk, then the kernel populates the page cache with the data so that any subsequent reads can access that data from cache. In case of write operations, there are three strategies:

1. **no-write**: a write operation against a page in cache would written directly to disk, invalidating the cached data.

2. **write-through**: a write operation will update both in-memory cache and on-disk file.

3. **write-back**: write operation happens directly into the page cache and the corresponding changes are not immediately written to disk. The written-to pages in the page cache are marked as *dirty* (hence dirty pages) and are added to a dirty list. Periodically, pages in the dirty list are written back to disk in a process called *writeback*.

We are more interested in the write-back strategy because this approach is used by FUSE (kernel driver). Dirty pages in the page cache needs to be written to disk. Dirty page write-back occurs in three situations:

1. When th amount of free memory in the system drops below a threshold value.
2. When dirty data grows older than a specified threshold, making sure dirty data does not remain dirty indefinitely.
3. When a user process invokes the `sync()` or `fsync()` system calls.

All of the above three tasks are performed by the group of flusher threads. First, flusher threads flush dirty data to disk when the amount of free memory in the system drops below a threshold value. This is done by a flusher thread calling a function `bdi_writeback_all`, which continues to write data to disk until following two conditions are true:

1. The specified number of pages has been written out; and
2. The amount of free memory is above the threshold.

Second, the flusher thread periodically wakes up and writes out old dirty data, thereby making sure no dirty pages remain in the page cache indefinitely.

Prior to Linux 2.6, the kernel had two threads: `bdflush` and `kupdated` which did exactly what the current flusher threads do. The `bdflush` thread was responsible for the background writeback of dirty pages (when free memory was low), while `kupdated` was responsible for periodic writeback of dirty pages. In the 2.6 kernel, a `pdflush` thread was introduced which performed similarly to the current flusher threads. The main difference was that the number of `pdflush` threads was dynamic between two and eight, depending on the I/O load. The pdflush threads were not associated with any specific disk, but instead they were global to all disks. The downside of pdflush was that it can easily bottleneck on congested disks, starving other devices from getting service. Therefore, a per-spindle flushing method was introduced to improve performace. The flusher threads replaced the pdflush threads in the 2.6.32 kernel [7]. The major disadvantage in bdflush was that it consisted of one thread. This led to congestion during heavy page writeback where the single bdflush thread blocked on a single slow device. The 2.6.32 kernel solved this problem by enabling multiple flusher threads to exists where each thread individually flushes dirty pages to a disk, allowing different threads to flush data at different rates to different disks. This also introduced the concept of *per-backing device info* (BDI) structure which maintains the per-device (disk) information like dirty list, read ahead size, flags, and B.D.Mn.R and B.D.Mx.R which are discussed in the next section.

### 2.8.2 Page Cache Parameters

**Global Background Ratio (G.B.R):** The percentage of **Total Available Memory** filled with dirty pages at which the background kernel flusher threads wake up and start writing the dirty pages out. The processes that generate dirty pages are not throttled at this point. G.B.R can be changed by the user at `/proc/sys/vm/dirty_background_ratio`. By default this value is set to 10%.

**Global Dirty Ratio (G.D.R):** The percentage of **Total Available Memory** that can be filled with dirty pages before the system starts to throttle incoming writes. When the system gets to this point, all new I/O's get blocked and the dirty data is written to disk until the amount of dirty pages in the system falls below this G.D.R. This value can be changed by the user at `/proc/sys/vm/dirty_ratio`. By default this value is set to 20%.

| Parameter Name | Short Name | Type | Default Value |
|---|---|---|---|
| Global Background Ratio | G.B.R | Percentage | 10% |
| Global Dirty Ratio | G.D.R | Percentage | 20% |
| BDI Min Ratio | B.Mn.R | Percentage | 0% |
| BDI Max Ratio | B.Mx.R | Percentage | 100% |
| Global Dirty Threshold | G.D.T | Absolute Value | - |
| Global Background Threshold | G.B.T | Absolute Value | - |
| BDI Dirty Threshold | B.D.T | Absolute Value | - |
| BDI Background Threshold | B.B.T | Absolute Value | - |

Table 2.3: Parameter names, default values (if applicable), and their shortcut names used throughout this thesis.

**Global Background Threshold (G.B.T):** The absolute number of pages in the system that, when crossed, the background kernel flusher thread will start writing out the dirty data. This is obtained from the following formula:

$$G.B.T = TotalAvailableMemory \times G.B.R$$

**Global Dirty Threshold (G.D.T):** The absolute number of pages that can be filled with dirty pages before the system starts to throttle incoming writes. This is obtained from the following formula:

$$G.D.T = TotalAvailableMemory \times G.D.R$$

**BDI Min Ratio (B.Mn.R):** Generally, each device is given a part of the page cache that relates to its current average write-out speed in relation to the other devices. This parameter gives the minimum percentage of the G.D.T (page cache) that is available to the file system. This value can be changed by the user at `/sys/class/bdi/`$\langle bdi \rangle$`/min_ratio` after the mount, where $\langle$*bdi*$\rangle$ is either a device number for block devices, or the value of st_dev on non-block-based file systems which set their own BDI information (e.g., a *fuse* file system). By default this value is set to 0%.

**BDI Max Ratio (B.Mx.R):** The maximum percentage of the G.D.T that can be given to the file system (100% by default). This limits the particular file system to use no more than the given percentage of the G.D.T. It is useful in situations where we want to prevent one file system from consuming all or most of the page cache. This value can be changed by the user at `/sys/class/bdi/`$\langle bdi \rangle$`/min_ratio` after a mount.

**BDI Dirty Threshold (B.D.T):** The absolute number of pages that belong to write-back cache that can be allotted to a particular device. This is similar to the G.D.T but for a particular BDI device. As a system runs, $B.D.T$ fluctuates between the lower limit ($G.D.T \times B.Mn.R$) and the upper limit ($G.D.T \times B.Mx.R$). Specifically, $B.D.T$ is computed using the following formula:

$$\begin{aligned}
B.D.T_{min} &= G.D.T \times B.Mn.R \\
B.D.T_{max} &= G.D.T \times B.Mx.R \\
B.D.T_{desired} &= G.D.T \times (100 - \sum_{bdi} B.Mn.R_{bdi}) \times WriteOutRatio \\
B.D.T &= min(B.D.T_{min} + B.D.T_{desired}, B.D.T_{max})
\end{aligned}$$

where $WriteOutRatio$ is the fraction of write-outs completed by the particular BDI device to that of the Global write-outs in the system, which is updated after every page is written to the device. The sum ($\sum$) in the formula is the minimum amount of page cache space guaranteed to every BDI. Only the remaining part ($100 - \sum$) is proportionally shared between BDIs.

**BDI Background Threshold (B.B.T):** When the absolute number of pages which are a percentage of G.D.T is crossed, then the background kernel flusher thread starts writing out the data. This is similar to the G.B.T but for a particular file system using BDI. This is obtained from the following formula:

$$B.B.T = B.D.T \times \frac{G.B.T}{G.D.T}$$

**NR_FILE_DIRTY:** The total number of pages in the system that are dirty. This parameter is incremented/decremented by the VFS (page cache).

**NR_WRITEBACK:** The total number of pages in the system that are currently under write-back. This parameter is incremented/decremented by the VFS (page cache).

**BDI_RECLAIMABLE:** The total number of pages belonging to all the BDI devices that are dirty. A file system that supports BDI is responsible for incrementing/decrementing the values of this parameter.

**BDI_WRITEBACK:** The total number of pages belonging to all the BDI devices that are currently under write-back. A file system that supports BDI is responsible for incrementing/decrementing the values for this parameter.

### 2.8.3 Write Back Cache Code Flow

Next, we explain the flow of write-back cache code within FUSE, when a process calls the write() system call. For every page within the I/O that is submitted, the following three events are performed in order:

**(1).** The page is marked as dirty and global parameters `NR_FILE_DIRTY`, `BDI_RECLAIMABLE`, `NR_WRITEBACK`, `BDI_WRITEBACK` are incremented accordingly.

**(2).** The `balance_dirty_pages_ratelimited()` function is called; it checks whether the task that is generating dirty pages needs to be throttled (paused) or not. Every task has a `rate_limit` assigned to it, which is initially set to 32 pages (by default). This function ensures that a task that is dirtying the pages never crosses the `rate_limit` threshold. As soon as task crosses that limit `balance_dirty_pages` is called. The reason to not call `balance_dirty_pages` directly (for every page) is that this operation is costly and calling that function for every page adds a lot of computational overhead.

**(3).** The `balance_dirty_pages()` function is the heart of the write-back cache, as it is the place where write-back and throttling is done. The following are the main steps being executed as part of this function:

- G.D.T, G.B.T, B.D.T, and B.B.T are calculated as described above. Since we are only interested in per-BDI file systems (as FUSE supports BDIs), BDI dirty pages are calculated as following:

$$bdi\_dirty = BDI\_RECLAIMABLE + BDI\_WRITEBACK$$

- If file systems initialize the BDI (during mount) with the `BDI_STRICT_LIMITS` flag, then the total number of `bdi_dirty_pages` should be under the `bdi_setpoint` which is calculated as follows:

$$bdi\_setpoint = \frac{(B.D.T + B.B.T)}{2}$$
$$bdi\_dirty \leq bdi\_setpoint$$

  If the above condition is true, then the task is not paused. The `rate_limit` parameter gives the number of pages this task can dirty before checking this condition again. Thus, it is calculated as follows:

$$nr\_dirtied\_paused = \sqrt{B.D.T - bdi\_dirty}$$

- If the above condition fails, then the flusher thread is woken up (if not already in progress) and dirty data is flushed.

- The task may be paused at this point depending on the `task_rate_limit` and `pages_dirtied` as follows:

$$pause = \frac{pages\_dirtied}{task\_rate\_limit}$$

  where pages_dirtied is the number of pages the task has dirtied from the last pause time to the current time. And, `task_rate_limit` is the dirty throttle rate which depends on the write bandwidth of the device. There is a minimum pause and a maximum pause time that a task can pause itself.

- Before exiting this function, a final condition is checked: whether the global dirty pages in the system have crossed the G.B.T. If so, the flusher thread is woken up (if not already in progress).

### 2.8.4 FUSE write-back and max write

The default write behavior of FUSE is synchronous and only 4KB of data is sent to the user daemon for writing (without big_writes). 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. This becomes a problem when the userspace uses some slower device as a storage back-end. The solution FUSE implemented was to make FUSE's page cache support a write-back policy and then make writes asynchronous. The default B.Mx.R in FUSE is set (hardcoded) to 1% (hardcoded). On a system with RAM size equal to 4GB, B.Mx.R is set to 1%, G.D.R is set to 20%, and G.B.R is set to 10%. Then, G.D.T will be 200,000 pages, G.B.T will be 100,000 pages; B.D.T and B.B.T will be 2,000 and 1,000 pages, respectively. Finally, `bdi_setpoint` will be 1,500 4KB pages, and thus equal to 6MB. This explains the low page cache size used in FUSE file systems. The reason that FUSE sets a low B.Mx.R limit (1%) is in case a FUSE file system running in user-space stalls for any reason; in that case, the dirty pages belonging to this FUSE file system would be stuck with in the page cache [22]. But B.Mn.R value can be safely increased if the daemon is trusted and running in a trusted environment (e.g., nobody can accidentally suspend it with SIG_STOP). With the write-back cache option, file data can be pushed to the user daemon in larger chunks using `max_write`, whose current limit is 32 pages. This many pages can be filled in each FUSE write request that is sent to the user daemon.

Table 2.4: Library Options

| Option | Explanation | Default Value |
|---|---|---|
| -h, –help | Prints help for this group of options | 0 (Don't Print) |
| -d, debug | Debug - log all the operations | 0 (Not Enabled) |
| hard_remove | Remove the file on unlink instead of creating a hidden file | 0 (Not Enabled) |
| use_ino | Honor the st_ino field in kernel functions getattr and fill_dir | 0 (Not Enabled) |
| readdir_ino | If use_ino option is not given, still try to fill in the d_ino field in readdir | 0 (Not Enabled) |
| direct_io | Enable Direct I/O (bypassing page cache) | 0 (Not Enabled) |
| kernel_cache | Disables flushing the cache of the file contents on every open | 0 (False) |
| auto_cache | Enable automatic flushing of data cache on open | 0 (False) |
| umask | Override the permission bits in st_mode set by the file system, given in octal | 0 (False) |
| uid | Override the st_uid field set by the file system with argumenet passed | 0 (False) |
| gid | Override the st_gid field set by the file system with argument passed | 0 (False) |
| entry_timeout | The length of time in seconds that name lookups will be cached | 1.0 secs |
| attr_timeout | The length of time in seconds that file/directory attributes are cached | 1.0 secs |
| ac_attr_timeout | The length of time that file attributes are cached, when checking if auto_cache should flush data on open | 0.0 secs |
| negative_timeout | The length of time in seconds that a negative lookup will be cached | 0.0 secs |
| noforget | Never forget cached inodes | 0 (means False) |
| remember | Remember cached inodes for that many seconds | 0 secs |
| intr | Allow requests to be interrupted | 0 (Disabled) |
| intr_signal | Which signal number to send to the file system when a request is interrupted | SIGUSR1 |
| modules | Names of modules to push onto the file system stack | NULL |

## 2.9 Miscellaneous Options

This section shows the various mount options (arguments) present in FUSE library that can be utilized (passed as arguments) by the file system owners during mount. Table 2.4 shows the various options/optimizations, their meaning, and their default values provided by the high-level API. These optimizations are implemented in the high-level API layer and they are not communicated to kernel driver. Only file systems implemented using the high-level API can utilize these options; others (using low-level) need to implement these options explicitly by themselves. Table 2.5 shows the various optimizations/options, their meaning, and their default values provided by the low-level API. These optimizations are implemented in the low-level API layer and are communicated with kernel driver during the INIT request processing. These options are available to file systems implemented by high-level and low-level FUSE API's.

Table 2.5: Low-Level Options

| **Option** | **Explanation** | **Default Value** |
|---|---|---|
| -h, –help | Prints help for this group of options | 0 (Don't print) |
| -d, debug | Debug | 0 (Disabled) |
| -V, –version | Print FUSE version number | 0 (Not printed) |
| allow_root | File access is limited to the user mounting the file system and root | 0 (Disabled) |
| max_write | Maximum size of write requests in bytes | 131072 bytes |
| max_readahead | Maximum readahead | 131072 bytes |
| max_background | Number of maximum background requests | 12 requests |
| congestion_threshold | Kernel's congestion threshold for background requests | 9 requests |
| async_read | Perform reads asynchronously | 1 (Enabled) |
| sync_read | Perform reads synchronously | 0 (Disabled) |
| atomic_o_trunc | Enable atomic open and truncate support | 0 (False) |
| no_remote_lock | Disable remote file locking | 0 (False) |
| no_remote_flock | Disable remote file locking (BSD) | 0 (False) |
| no_remote_posix_lock | Disable remove file locking (POSIX) | 0 (False) |
| big_writes | Enable larger than 4KB writes | 0 (False) |
| splice_write | Use splice to write to the FUSE device | 0 (False) |
| splice_move | Move data while splicing to the FUSE device | 0 (False) |
| splice_read | Use splice to read from the FUSE device | 0 (False) |
| auto_inval_data | Use automatic kernel cache invalidation logic | 0 (False) |
| readdirplus | Control readdirplus use (yes\|no\|auto) | auto |
| async_dio | Asynchronous direct I/O | 0 (False) |
| writeback_cache | Enable asynchronous, buffered writes | 0 (False) |
| time_gran | Time granularity in nano-seconds | 0 secs |
| clone_fd | Separate /dev/fuse device file descriptor for each processing thread | 0 (False) |

# Chapter 3

# Implementations

To study FUSE's resource utilization, performance, and overall behavior, we developed a simple stackable pass-through 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 plans to develop an efficient FUSE-based file system. We first describe the implementation of Stackfs in Section 3.1. Then, in Section 3.2, we describe the performance statistics that we extracted from the FUSE kernel and user library using the newly added instrumentation. And finally, in Section 3.3 we show code changes of various components that were developed and implemented as part of this research project.

## 3.1 Stackfs

Stackfs is a Stackable user-space file system implemented using the FUSE framework; Stackfs layers on top of an Ext4 file system in our experiments (but it can stack on top of any other file system). Stackfs passes FUSE requests unmodified directly to the underlying file system (Ext4). The reason we developed Stackfs was twofold:

1. The majority of existing FUSE-based file systems are stackable (i.e., deployed on top of other, often in-kernel file systems). Therefore, evaluation results obtained via Stackfs are applicable to the largest class of user-space file systems.

2. We wanted to add as little overhead as possible, so as 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 complex 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. Next 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 required bookkeeping information, our inode stores the path to the underlying file, its inode number, and a reference counter. The path is used to open the underlying file when an OPEN request for a Stackfs file arrives. We maintain reference counts to avoid creating a new inode if we need to lookup a path that has already been looked up; for that, we maintain all inodes that have been looked up so far in a hash table indexed by the underlying file system inode number. Below we have the structure for Stackfs' inode. This structure is very similar to the `struct node` implemented by the FUSE library to support the high level API.

```
struct stackFS_inode {
        char *path;
        ino_t ino; /* underlying file system inode number */
        uint64_t nlookup;
        ...
};
```

**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 type-casting), which is guaranteed to be unique. This makes the inode number space sparse but 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 hard-links) 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.

Unlike our implementation, FUSE high-level library maintains two hash tables: one for mapping FUSE inode numbers to FUSE inodes and another for mapping FUSE inodes to file paths. In our implementation, however, we maintain only one hash table which maps StackFS inodes to their underlying file system file paths. Our implementation uses the StackFS inode structure memory address as the inode number; this simplifies the mapping of StackFS inode numbers to StackFS inodes.

**Session information.** The low-level API allows user-space file systems to store private information for each FUSE session/connection. This important data is then made available to any request that a user space file system serves. We are storing a reference to the hash table and the root node in this structure. The hash table is referenced every time a new inode is created or looked up. The root node contains the path (mount point) of the underlying file system which is passed during the mount, so this path is prepended to all path conversions and used by Stackfs.

**Directories.** In Stackfs we use directory handles for accessing the directories similar to the file handles for files. A directory handle stores the directory stream pointer for the (opened) directory, the current offset within the directory (useful for `readdir`), and information about files within the directory. This handle is useful in all directory operations; importantantly, this handle can be stored as part of the `struct fuse_file_info` which stored information about open files/directories. Below we describe the structure for Stackfs's directory handle. This structure is similar to the `struct fuse_dh` which is implemented by the Fuse library to support the high Level API.

```
struct stackFS_dirptr {
        DIR *dp;
        struct dirent *entry;
        off_t offset;
};
```

**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; it is also useful for any additional functionality (e.g., encryption, compression). The file descriptor is deallocated when the file is closed.

We made our code publicly available at:

- Repo: `https://github.com/sbu-fsl/fuse-stackfs.git`
- Branch: `master`
- Path (within above repo): ⟨`GitLocation`⟩`/StackFS_LowLevel/`

## 3.2 Performance Statistics and Traces

After starting this project, we quickly realized that 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 sufficient to capture the worst latencies in typical FUSE setups. (This technique efficiently records a $log_2$ latency histogram was introduced first in OSprof [30].)

We then added four such arrays to FUSE: the first three arrays are in kernel (in `fuse_conn` structure) and are assigned to each of FUSE's three main queues (background, pending, and processing); the fourth array is in user space (in `fuse_session` structure) and tracks the time the daemon needs to process a request. The total memory size of all four arrays is only 48KiB; and only a few CPU instructions are necessary to update values in the array. We added functions to FUSE's library and kernel, to capture the timings of requests and also to update the four arrays. These can be used by user-space file-system developers to track the latencies at different parts of the code flow.

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). We added three new files to these `fusectl` directories directories to export statistics from the in-kernel arrays: `background_queue_requests_timings`, `pending_queue_requests_timings`, and `processing_queue_requests_timings`. To export the user-level array we added a `SIGUSR1` signal handler to the daemon. When triggered, the handler prints the array to a log file specified during the daemon's start. This method allows us to retrieve the statistics we want at any desired frequency. The statistics captured have no measurable overhead on FUSE's performance and are the primary source of information we used to study FUSE's performance.

### 3.2.1 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 [17] to add over 30 tracepoints to FUSE's kernel module. These new tracepoints are mainly

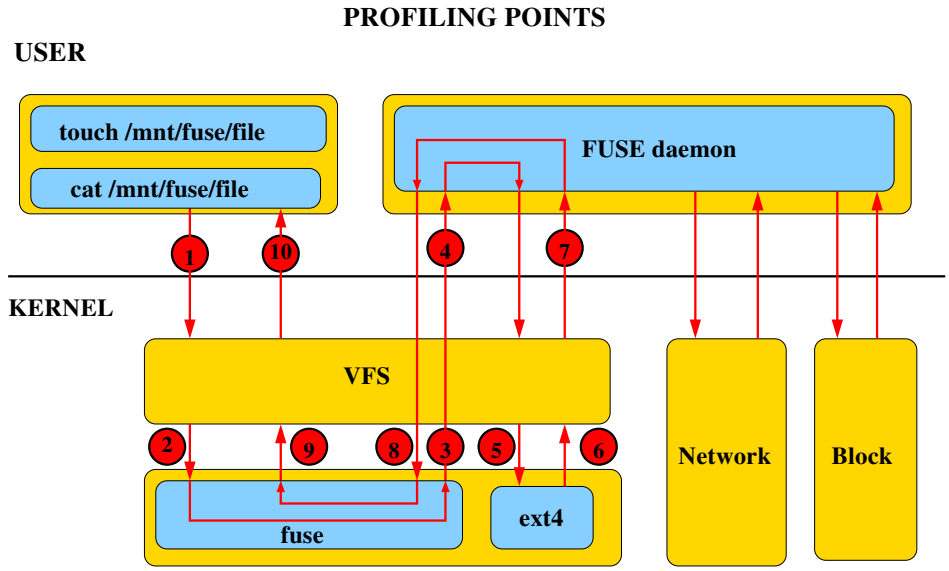


Figure 3.1: Locations of tracepoints throughout the flow of FUSE requests.

to monitor the creation of requests during the complex writeback logic, track the amount of data being read/written per request, and track metadata operations (to know how often they get generated). Figure 3.1 shows different trace points that we tracked during a normal FUSE request flow. Tracing helped us learn how fast do the queues grow and shrink 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. We made our code publicly available at:

1. Fuse Kernel Instrumentation:
   - Repo: `https://github.com/sbu-fsl/fuse-kernel-instrumentation.git`
   - Branch: `master`
2. Fuse Library Instrumentation:
   - Repo: `https://github.com/sbu-fsl/fuse-library-instrumentation.git`
   - Branch: `master`

## 3.3 Development Effort

Table 3.1 shows code changes of components that were developed and implemented as part of this research project.

| Module | Language | Files | Insertions | Deletions |
|---|---|---|---|---|
| Linux Kernel Traces and Performance Stats | C | 11 | 2181 | 934 |
| | CMake | 2 | 23 | 3 |
| | C Header | 5 | 1094 | 66 |
| | **Subtotal:** | **18** | **3298** | **1003** |
| FUSE Library Performance Stats | C | 4 | 219 | 31 |
| | Version Script | 1 | 6 | 1 |
| | C Header | 2 | 41 | 2 |
| | **Subtotal:** | **7** | **266** | **34** |
| StackFS, Workloads, Automated Scripts and Plotting Scripts | C | 1 | 1483 | 319 |
| | C++ | 15 | 3249 | 398 |
| | Shell | 20 | 2566 | 241 |
| | WML | 271 | 22851 | 2132 |
| | Octave | 38 | 6141 | 514 |
| | CMake | 2 | 66 | 23 |
| | **Subtotal:** | **347** | **36356** | **3627** |
| **TOTAL:** | | **372** | **39920** | **4664** |

Table 3.1: Development effort in project. WML is Filebench's Workload Model Language.

# Chapter 4

# Methodology

FUSE has evolved significantly over the years and added several useful optimizations: a writeback cache, zero-copy via splicing, and multi-threading. In our experience, some in the storage community tend to pre-judge FUSE's performance—assuming it is poor—mainly due to not having enough information about the improvements FUSE has 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. In this chapter we detail our methodology, starting from the description of FUSE configurations in Section 4.1, proceed to the list of workloads in Section 4.2, and finish by presenting our testbed in Section 4.3.

## 4.1 Configurations

To demonstrate the evolution of FUSE's performance, we picked two configurations on opposite sides of the spectrum:

1. The *basic* configuration (called *StackfsBase*) with no major FUSE optimizations,
2. 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. the writeback cache is turned on;
2. the maximum size of a single FUSE request was increased from 4KiB to 128KiB (`max_write` parameter), to allow larger data transfers;
3. the user daemon runs in the multi-threaded mode; and
4. splicing is activated for all operations (using the `splice_read`, `splice_write`, and `splice_move` parameters).

We left all other parameters at their default values in both configurations: for example, `readahead` defaults to 32 pages and `async_read` is on.

| Workload Name | Description |
|---|---|
| rnd-rd-*N*th-1f | *N* threads (1, 32) randomly read from a single preallocated 60GB file. |
| rnd-wr-*N*th-1f | *N* threads (1, 32) randomly write to a single preallocated 60GB file. |
| seq-rd-*N*th-1f | *N* threads (1, 32) sequentially read from a single preallocated 60GB file. |
| seq-wr-1th-1f | Single thread creates and sequentially writes a new 60GB file. |
| seq-rd-32th-32f | 32 threads sequentially read 32 preallocated 2GB files. Each thread reads its own file. |
| seq-wr-32th-32f | 32 threads sequentially write 32 new 2GB files. Each thread writes its own file. |
| files-cr-*N*th | *N* threads (1, 32) create 4 million 4KB files over many directories. |
| files-rd-*N*th | *N* threads (1, 32) read from 1 million preallocated 4KB files over many directories. |
| files-del-*N*th | *N* threads (1, 32) delete 4 million of preallocated 4KB files over many directories. |
| web-server | Web-server workload emulated by Filebench. Scaled up to 1.25 million files. |
| mail-server | Mail-server workload emulated by Filebench. Scaled up to 1.5 million files. |
| file-server | File-server workload emulated by Filebench. Scaled up to 200,000 files. |

Table 4.1: Description of workloads. For data-intensive workloads, we experimented with 4KB, 32KB, 128KB, and 1MB I/O sizes. We picked data-set sizes so that both cached and non-cached data are exercised.

## 4.2 Workloads

To stress different modes of FUSE operation and conduct a 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 4.1 describes all workloads that we employed in the evaluation. To simplify the identification of workloads in the text, we use the following short 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. Single-threaded workloads represent the most basic workloads while 32-threads is enough to load all cores of the CPU in our system. In this thesis 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. The resulting runtimes varied between 8 and 20 minutes across all experiments. Note that because for some workloads, SSDs are orders of magnitude faster than HDDs, we selected a larger amount of work for our SSD experiments than HDD-based ones. We used Filebench [19, 52] to generate all workloads. To encourage reuse and reproducibility, we released the Filebench personality files along with raw experimental results:

- Repo: `https://github.com/sbu-fsl/fuse-stackfs.git`
- Branch: `master`

## 4.3 Experimental Setup

FUSE's performance impact depends heavily on the speed of the underlying storage: faster devices highlight FUSE's own overheads. We therefore experimented with two common storage devices of different speeds: 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 a 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 the CentOS 7 distribution with the Linux kernel upgraded to v4.1.13 and FUSE library commit #386b1b (contains latest features as of this writing).

We used Ext4 [18] 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 [11]: we initialized all inodes during the formatting rather than asynchronously immediately after mounting the file system for the first time. We did see a high standard deviation in our experiments with the default lazy inode initialization mode; selecting this option has reduced the standard deviation. In either case, the measured standard deviations in our experiments were less than 2% for all workloads except for three workloads: `seq-rd-1th-1f` (6%), `files-rd-32th` (7%), and `mail-server` (7%).

# Chapter 5

# Evaluation

We believe that for many researchers and practitioners, the main benefit of this thesis is the characterization of FUSE's performance for different workloads, configurations, and hardware. For many, FUSE's framework is just a practical *tool* to build a real product or a prototype, not a research focus. Therefore, to present our results more effectively, we organized our evaluation in two parts. Section 5.1 overviews our extensive evaluation results. 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. We also measured the CPU utilization during the experiments. As detailed in Chapter 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 5.1 shows absolute throughputs for Ext4 and relative performance for two Stackfs configurations for both HDD and SSD. We also calculated the relative CPU utilization of Stackfs vs. Ext4 for each workload. Table 5.1 shows the results for the CPU utilization in seconds separately for User and Kernel times, for two Stackfs configurations, and for both the HDD and SSD.

For better clarity we categorized the results (from Table 5.1) by Stackfs's performance difference into five classes:

1. The *Blue* class (marked with $^{@}$) indicates that the performance actually improved;
2. The *Green* class (marked with $^{+}$) indicates that the performance degraded by less than 5%;
3. The *Yellow* class ($^{*}$) includes results with the performance degradation in the 5–25% range;
4. The *Orange* class ($^{\#}$) indicates that the performance degradation is between 25–50%;
5. And finally, the *Red* class ($^{!}$) is for when performance decreased by more than 50%.

Similarly, we categorized the results (from Table 5.1) by Stackfs's CPU utilization (compared to Ext4) into three classes:

1. The *Green* class indicates that CPU utilization is less than 2×;
2. The *Orange* class indicates that CPU utilization is in the 2–50× range;

Table 5.1: List of workloads and corresponding performance results. Stackfs1 refers to StackfsBase and Stackf2 refers to StackfsOpt.

| # | Workload | I/O Size (KB) | HDD Results | | | SSD Results | | |
|---|---|---|---|---|---|---|---|---|
| | | | EXT4 (ops/s) | Stackfs1 (%Diff) | Stackfs2 (%Diff) | EXT4 (ops/s) | Stackfs1 (%Diff) | Stackfs2 (%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-32f | 4 | 11141 | - 36.9$^{\#}$ | - 26.9$^{\#}$ | 32855 | - 0.1$^{+}$ | - 0.16$^{+}$ |
| 6 | | 32 | 1491 | - 41.5$^{\#}$ | - 30.3$^{\#}$ | 4202 | - 0.07$^{+}$ | - 1.8$^{+}$ |
| 7 | | 128 | 371 | - 41.3$^{\#}$ | - 29.8$^{\#}$ | 1051 | - 0.1$^{+}$ | - 0.2$^{+}$ |
| 8 | | 1024 | 46 | - 41.0$^{\#}$ | - 28.3$^{\#}$ | 131 | - 0.03$^{+}$ | - 2.1$^{+}$ |
| 9 | seq-rd-32th-1f | 4 | 1228400 | - 2.4$^{+}$ | - 3.0$^{+}$ | 973450 | + 0.02$^{@}$ | + 2.1$^{@}$ |
| 10 | | 32 | 153480 | - 2.4$^{+}$ | - 4.1$^{+}$ | 121410 | + 0.7$^{@}$ | + 2.2$^{@}$ |
| 11 | | 128 | 38443 | - 2.6$^{+}$ | - 4.4$^{+}$ | 30338 | + 1.5$^{@}$ | + 1.97$^{@}$ |
| 12 | | 1024 | 4805 | - 2.5$^{+}$ | - 4.0$^{+}$ | 3814.50 | - 0.1$^{+}$ | - 0.4$^{+}$ |
| 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 | - 0.07$^{+}$ | + 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 5.2: List of workloads and corresponding CPU utilization in secs. FUSE1 refers to StackfsBase and FUSE2 refers to StackfsOpt.

| S.No | Workloads | I/O Size (KB) | HDD Results (secs) | | | | | | SSD Results (secs) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | EXT4(secs) | | FUSE1(×) | | FUSE2(×) | | EXT4(secs) | | FUSE1(×) | | FUSE2(×) | |
| | | | usr | sys | usr | sys | usr | sys | usr | sys | usr | sys | usr | sys |
| 1 | | 4 | 8.5 | 47 | 1.4 | 2.3 | 1.8 | 2.6 | 8.5 | 48 | 1.4 | 2.3 | 1.8 | 2.7 |
| 2 | sq-rd- | 32 | 2.9 | 43 | 2.2 | 2.5 | 3.2 | 2.8 | 3 | 44 | 2 | 2.4 | 3.1 | 2.9 |
| 3 | 1th-1f | 128 | 1.9 | 42.6 | 2.5 | 2.4 | 4.2 | 2.8 | 2 | 43 | 2.4 | 2.3 | 4.0 | 2.8 |
| 4 | | 1024 | 0.6 | 42 | 5.8 | 2.4 | 10.3 | 2.7 | 0.7 | 43 | 5 | 2.3 | 9.0 | 2.8 |
| 5 | | 4 | 9 | 48 | 1.7 | 2.6 | 1.9 | 2.6 | 8 | 46 | 1.6 | 2.6 | 1.9 | 2.5 |
| 6 | sq-rd- | 32 | 3.4 | 44 | 2.4 | 2.6 | 3.2 | 2.6 | 2.5 | 42 | 2.4 | 2.7 | 3.6 | 2.6 |
| 7 | 32th-32f | 128 | 2.8 | 45 | 2.5 | 2.6 | 3.4 | 2.6 | 1.9 | 42 | 2.7 | 2.8 | 4.2 | 2.6 |
| 8 | | 1024 | 1.7 | 44 | 3.1 | 2.6 | 5.1 | 2.7 | 0.8 | 42 | 3.9 | 2.8 | 7.5 | 2.7 |
| 9 | | 4 | 476 | 1045 | 1.0 | 1.1 | 1.0 | 1.1 | 483 | 1054 | 1.0 | 1.1 | 1.0 | 1.1 |
| 10 | sq-rd- | 32 | 80 | 691 | 1.1 | 1.1 | 1.2 | 1.2 | 81 | 703 | 1.1 | 1.1 | 1.2 | 1.2 |
| 11 | 32th-1f | 128 | 41 | 722 | 1.1 | 1.1 | 1.4 | 1.2 | 41 | 715 | 1.2 | 1.1 | 1.5 | 1.2 |
| 12 | | 1024 | 11 | 1190 | 1.5 | 1.1 | 3.6 | 1.1 | 11 | 1200 | 1.5 | 1.0 | 3.6 | 1.1 |
| 13 | | 4 | 1.7 | 8.7 | 2.6 | 2.2 | 3.5 | 4.0 | 28 | 194 | 5.4 | 3.5 | 8.8 | 7.4 |
| 14 | rnd-rd- | 32 | 1.9 | 12.6 | 2.4 | 2.3 | 3.1 | 3.4 | 6.8 | 71 | 3.6 | 2.6 | 4.4 | 3.7 |
| 15 | 1th-1f | 128 | 1.5 | 17 | 2.3 | 2.7 | 2.8 | 2.9 | 2 | 44 | 2.9 | 2.4 | 3.7 | 2.6 |
| 16 | | 1024 | 0.9 | 44 | 4.6 | 2.6 | 8.1 | 2.8 | 0.8 | 43 | 5.2 | 2.5 | 9.2 | 2.8 |
| 17 | | 4 | 0.8 | 4.5 | 4.5 | 3.9 | 5.6 | 5.9 | 34 | 272 | 2.7 | 1.9 | 8.1 | 5.2 |
| 18 | rnd-rd- | 32 | 1 | 8 | 4 | 3.4 | 4.5 | 4.4 | 4.9 | 60 | 3.6 | 2.8 | 4.8 | 3.5 |
| 19 | 32th-1f | 128 | 1.2 | 17 | 2.8 | 2.6 | 3.2 | 2.7 | 1.9 | 43 | 3.2 | 2.6 | 3.7 | 2.4 |
| 20 | | 1024 | 1.7 | 45 | 3.1 | 2.6 | 4.4 | 2.5 | 0.8 | 44 | 4.2 | 2.7 | 7.2 | 2.5 |
| 21 | | 4 | 10 | 108 | 6.6 | 3.8 | 1.7 | 4.4 | 9 | 104 | 6.6 | 3.8 | 1.7 | 4.6 |
| 22 | sq-wr- | 32 | 1.5 | 81 | 30.4 | 4.6 | 4.8 | 5.4 | 1.6 | 83 | 29.4 | 4.5 | 4.3 | 5.2 |
| 23 | 1th-1f | 128 | 0.8 | 80 | 57.9 | 4.6 | 8.1 | 5.4 | 0.9 | 83 | 49 | 4.5 | 7.1 | 5.2 |
| 24 | | 1024 | 0.5 | 81 | 90.3 | 4.6 | 12.9 | 5.4 | 0.5 | 83 | 85.4 | 4.4 | 12.0 | 5.2 |
| 25 | | 4 | 1.3 | 31 | 51.6 | 12.3 | 6.9 | 12.0 | 1.6 | 32.4 | 43.8 | 11.8 | 5.9 | 11.6 |
| 26 | sq-wr- | 32 | 0.6 | 28 | 69.4 | 12.4 | 13.1 | 12.7 | 0.7 | 28 | 61.7 | 12.5 | 12.3 | 13.0 |
| 27 | 32th-32f | 128 | 0.5 | 29.0 | 69.4 | 12.2 | 14.4 | 12.4 | 0.6 | 29 | 61.9 | 12.2 | 13.5 | 12.7 |
| 28 | | 1024 | 0.5 | 29 | 74.4 | 12.4 | 22.2 | 12.5 | 0.5 | 29 | 64.1 | 12.1 | 20.4 | 12.2 |
| 29 | | 4 | 1.4 | 10 | 4.2 | 3.3 | 8.4 | 15.8 | 1.9 | 86 | 31.4 | 4.8 | 66.3 | 14.8 |
| 30 | rd-wr- | 32 | 1.3 | 16 | 15.1 | 9.2 | 3.8 | 6.9 | 1.7 | 70 | 25.8 | 4.9 | 11.2 | 7.2 |
| 31 | 1th-1f | 128 | 1.4 | 30 | 28.4 | 10.1 | 5.3 | 10.3 | 0.7 | 53 | 57.6 | 6.2 | 9.7 | 6.9 |
| 32 | | 1024 | 0.7 | 33 | 55.8 | 9.8 | 9.6 | 10.6 | 0.5 | 51 | 78 | 6.3 | 13.1 | 7.2 |
| 33 | | 4 | 1.6 | 17 | 4.2 | 2.5 | 11.0 | 14.0 | 15 | 409 | 4.6 | 1.2 | 8.9 | 4.0 |
| 34 | rd-wr- | 32 | 1.4 | 16 | 15.2 | 9.5 | 5.2 | 9.0 | 1.9 | 103 | 23.9 | 3.5 | 10.3 | 5.2 |
| 35 | 32th-1f | 128 | 1.5 | 28 | 26.2 | 11.1 | 5.6 | 12.2 | 0.9 | 77 | 46.3 | 4.5 | 7.9 | 5.1 |
| 36 | | 1024 | 0.8 | 33 | 50.2 | 10 | 8.9 | 11.3 | 0.5 | 52 | 75.9 | 6.4 | 12.9 | 7.4 |
| 37 | files-cr-1th | 4 | 27 | 298 | 4.8 | 2.6 | 10.4 | 4.9 | 25.7 | 294.6 | 5.1 | 2.7 | 10.8 | 4.9 |
| 38 | files-cr-32th | 4 | 37 | 902 | 3.5 | 0.9 | 7.9 | 1.5 | 38 | 897 | 3.3 | 0.9 | 8.0 | 1.5 |
| 39 | files-rd-1th | 4 | 4.5 | 27 | 2.7 | 2.1 | 5.5 | 4.1 | 9 | 60 | 3.9 | 2.9 | 8.7 | 5.5 |
| 40 | files-rd-32th | 4 | 3 | 20 | 5 | 3.8 | 6.5 | 4.7 | 11 | 96 | 4.1 | 2.2 | 11.7 | 4.1 |
| 41 | files-del-1th | - | 8.6 | 85 | 4.0 | 2.3 | 7.6 | 3.4 | 15 | 167 | 4.3 | 2.3 | 8.3 | 3.5 |
| 42 | files-del-32th | - | 11 | 168 | 3.5 | 1.2 | 5.2 | 1.6 | 22 | 392 | 4.0 | 1.1 | 5.3 | 1.4 |
| 43 | file-server | - | 5.5 | 50.4 | 3.8 | 3.0 | 6.3 | 3.3 | 14 | 147 | 4.3 | 2.9 | 9.6 | 3.5 |
| 44 | mail-server | - | 4.1 | 41 | 3.9 | 2.4 | 5.3 | 2.9 | 31 | 357 | 3.2 | 1.8 | 8.5 | 2.9 |
| 45 | web-server | - | 17 | 26 | 2.6 | 3.5 | 2.4 | 4.2 | 46 | 193 | 3.3 | 2.9 | 9.2 | 5.5 |

3. And finally, the *Red* class indicates that CPU utilization is 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 performance 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` and `files-cr-32th`.

***Observation 5*** The results indicate that 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, the all write-intensive workloads (sequential and random) [rows #21–36] are within the *Green* and *Blue* classes for both HDD and SSD.

***Observation 7*** The performance of sequential read [rows #1–12] are well within the *Green*, *Blue* 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; this behaviour is seen in 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.

Similarly, below we list our main observations drawn from the CPU utilization results using Table 5.1.

***Observation 1*** The relative increase in the CPU utilization varied across workloads, devices, and FUSE configurations from 1.0× for `sq-re-32th-1f` [row #21] to 90.3× `sq-wr-1th-1f` [row #4].

***Observation 2*** For many workloads, FUSE's optimizations reduce CPU utilization significantly for user times while kernel times remained the same. One exception is [row #9] `rd-wr-1th-1f`, where CPU utilization increased with FUSE's optimizations.

***Observation 3*** With an increase in the I/O size among individual workloads, there is an increase in the CPU utilization. This behaviour is seen especially with user times rather than kernel times [rows #1–36].

***Observation 4*** There is not much difference between the CPU utilization results for most of the workloads among HDD's and SSD's setup. For example, the user and kernel CPU utilization results are almost indentical for `sq-re-32th-1f` [rows #21–24] on HDD and SSD.

***Observation 5*** Among all the workloads, only user CPU utilization belongs to the Red class for both StackfsBase and StackfsOpt[rows #3–8], [row #9], [rows #11–12], [row #16]. That is, none of the kernel CPU utilization falls below the Red class.

## 5.2 Analysis

We analyzed FUSE behavior in details using our instrumentation and present main findings here. We follow the order of workloads in Table 5.1.

**Read Workloads**

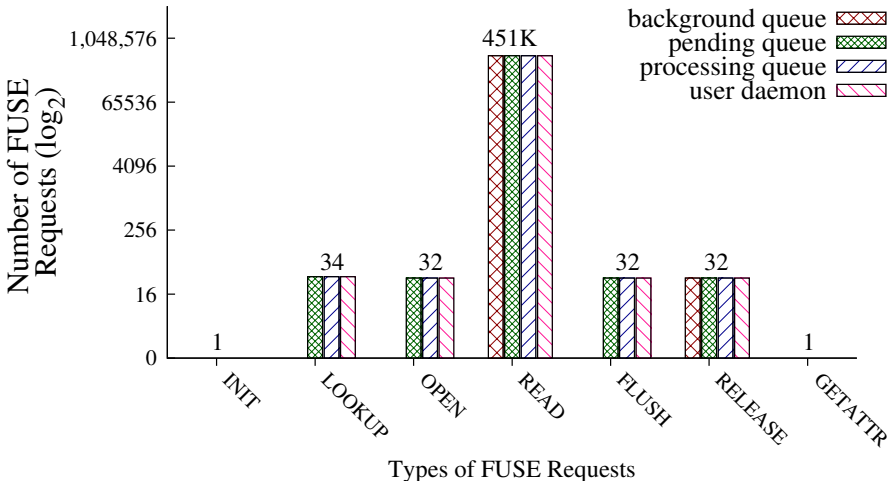


Figure 5.1: Different types and number of requests generated by StackfsBase on SSD during the `seq-rd-32th-32f` workload, from left to right, in their order of generation.

Figure 5.1 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 5.1 also shows the breakdown of requests across queues. By default, READ, RELEASE, and INIT are asynchronous requests. Therefore, they are added to background queue first, whereas all other requests are synchronous and are added to pending queue directly. In read-intensive workloads, only READ requests are generated in large quantities compared to other request types. Therefore we consider only READ requests when we discuss each workload in detail.

For all the read-intensive workloads [rows #1–20], the CPU utilization of StackfsBase and StackfsOpt fall under *Green* and *Orange* class. Moreover, the CPU user time is more (when compared to system time) because the file system daemon runs in user space.

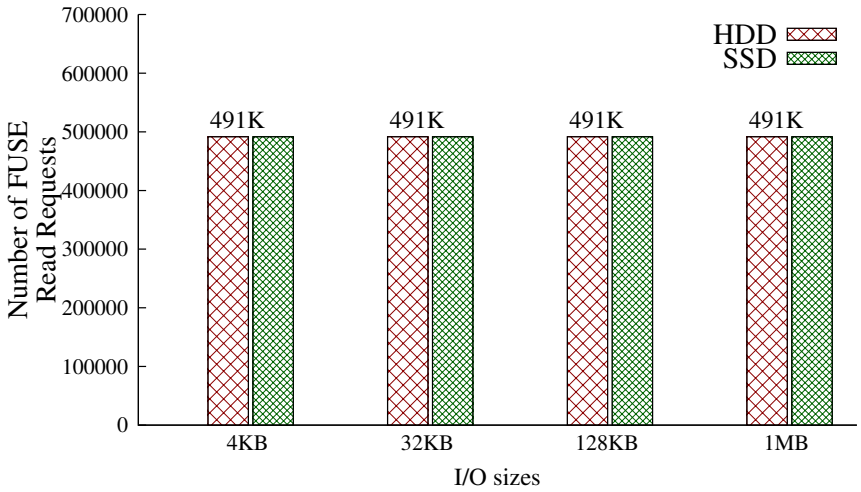


Figure 5.2: Total number of READ requests that were generated by StackfsBase on HDD and SSD for the `seq-rd-1th-1f` workload.

**Sequential Read using 1 thread on 1 file** Figure 5.2 shows the total number of READ requests that StackfsBase generated during the whole experiment for different I/O sizes for HDD and SSD. Surprisingly, the number of requests remained approximately the same across the I/O sizes. 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 in this experiment was also approximately same across 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. 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. However, the results were still 26–30% farther from Ext4 performance. When

| # | Workload | I/O Size (KB) | HDD Results | | | |
|---|---|---|---|---|---|---|
| | | | EXT4 (ops/s) | Stackfs1 (%Diff) | Stackfs2 (%Diff) | Stackfs3 (%Diff) |
| 5 | seq-rd-32th-32f | 4 | 11141 | - 36.9[#] | - 26.9[#] | + 0.18[@] |
| 6 | | 32 | 1491 | - 41.5[#] | - 30.3[#] | - 1.0[+] |
| 7 | | 128 | 371 | - 41.3[#] | - 29.8[#] | - 2.7[+] |
| 8 | | 1024 | 46 | - 41.0[#] | - 28.3[#] | - 2.0[+] |

Table 5.3: List of workloads and corresponding performance results. Stackfs1 refers to StackfsBase, Stackfs2 refers to StackfsOpt, and Stackfs3 referes to StackfsBase with increased background_queue limit.

we investigated further, we found that in case of HDD and StackfsOpt, FUSE daemon was bound by the max_background value (default is 12); at any given time, only 12 user deamons (threads) were invoked. So we increased that limit to 100 and reran the experiments. Table 5.3 shows the results, which demonstrates that StackfsOpt was now 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 request. 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.

### Write Workloads

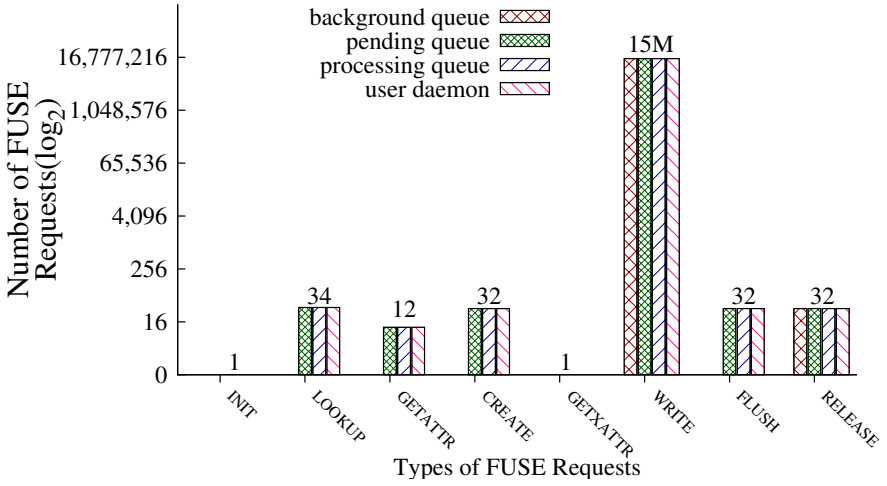


Figure 5.3: Different types of requests that were generated by StackfsBase on SSD for the `seq-wr-32th-32f` workload, from left to right in their order of generation.

We now discuss the behavior of StackfsBase and StackfsOpt in all write workloads listed in Table 5.1 [rows #21–36]. Figure 5.3 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 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. For all write-intensive workloads [rows #21–36], the CPU utilization of StackfsBase and StackfsOpt falls under *Red* and *Orange* classes. Moreover, for most of the cases, CPU user time is under *Red* (when compared to system time) for StackfsBase, because the file system daemon running in user space.

**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).

**Sequential Write using 32 threads on 32 files** Performance trends are similar to `seq-wr-1th-1f` but even the unoptimized StackfsBase performed significantly better (up to –2.7% and –0.07% degradation for HDD and SSD, respectively). This is explained by the fact that 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** The performance degradation caused by StackfsBase and StackfsOpt was low on HDD for all I/O sizes (not more than –1.3%) because the random write performance of Ext4 on HDD is low—between 79 and 1074 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 not visible.

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` so the same analysis applies.

### Metadata Workloads

We now discuss the behavior of StackfsBase and StackfsOpt in all metadata-intensive micro-workloads as listed in Table 5.1 [rows #37–42].

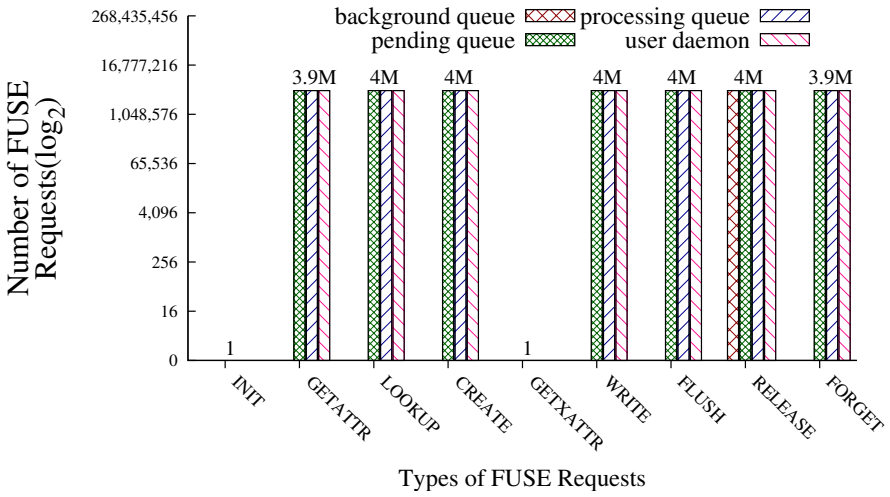


Figure 5.4: Different types of requests that were generated by StackfsBase on SSD for the `files-cr-1th` workload, from left to right in their order of generation.

**File creates** Figure 5.4 shows different types of requests that got generated during the `files-cr-Nth` runs. 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 Ext4 (30–46 thousand creates/sec) because small newly created inodes can be effectively cached in RAM. As a result, the overhead caused by the FUSE requests user-kernel communication explains the performance degradation.

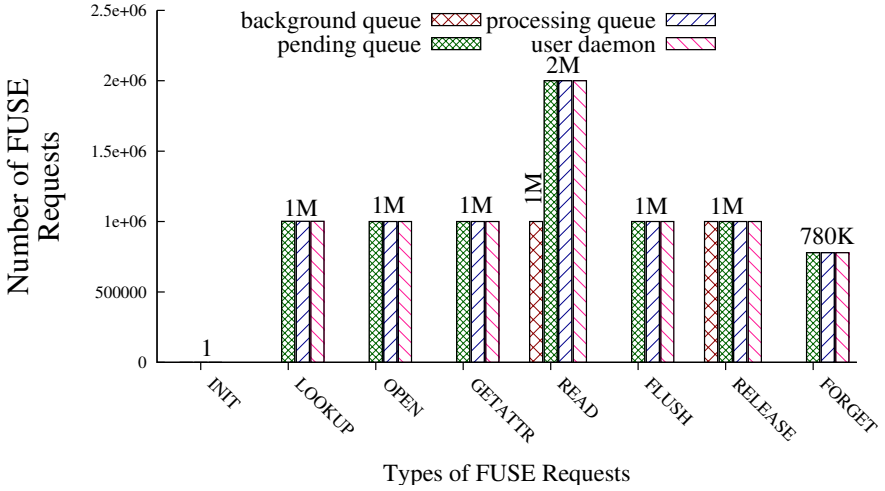


Figure 5.5: Different types of requests that were generated by StackfsBase on SSD for the `files-rd-1th` workload, from left to right in their order of generation.

**File Reads** Figure 5.5 shows different types of requests that got generated during the `files-rd-1th` workload. We classify this workload as metadata-intensive because it contains many small files (one million 4KB files) that are repeatedly opened and closed. Figure 5.5 shows that half of the READ requests went to the background queue and the other half directly to the pending queue. The reason 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 (instead of 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 and close to Ext4; on SSD, however, the relative degradation is high (–25%) because SSD is 12.5× faster than HDD (see Ext4 absolute throughput in Table 5.1). 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 per request. 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** Figure 5.6 shows different types of operations that got generated during the `files-del-1th` workloads. Every UNLINK request is followed by FORGET. Therefore, for every incoming delete re-

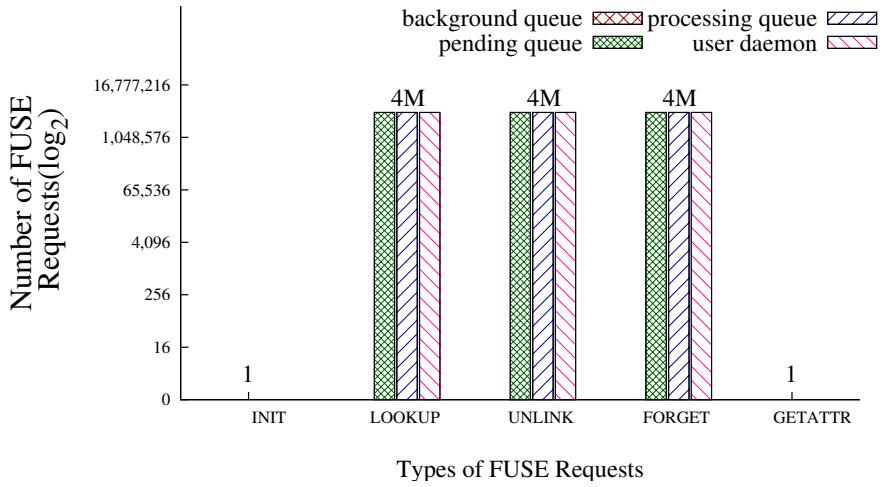


Figure 5.6: Different types of requests that were generated by StackfsBase on SSD for the `files-del-1th` workload, from left to right in their order of generation.

quest 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-`*`N`*`th` workloads and only add additional overhead for managing extra threads.

### Macro Server Workloads

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

**File Server** Figure 5.7 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 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 `max_write`, the number of WRITE requests 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.

**Mail Server** Figure 5.8 shows different types of operations that got generated during the `mail-server` workload. As with the `file-server` workload, many 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.

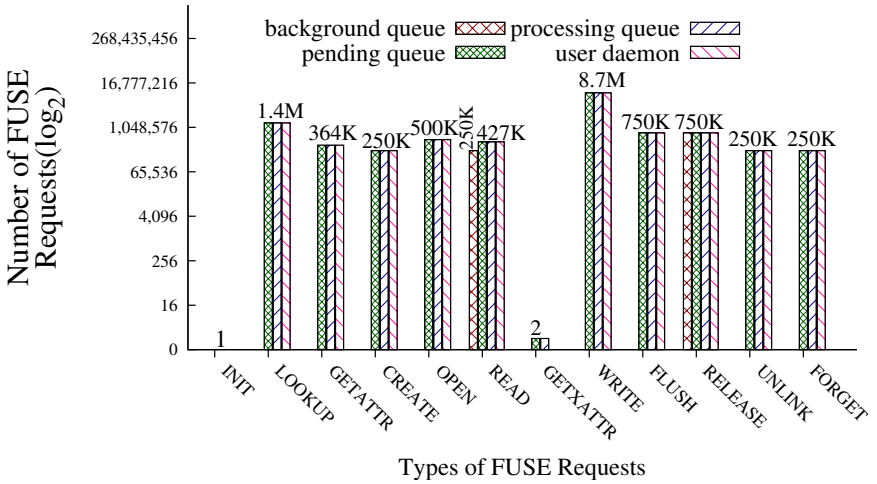


Figure 5.7: Different types of requests that were generated by StackfsBase on SSD for the `file-server` workload.

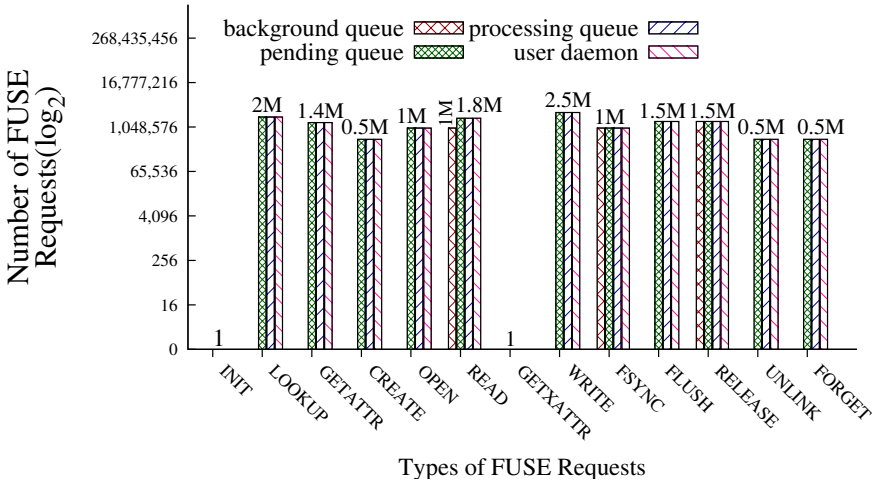


Figure 5.8: Different types of requests that were generated by StackfsBase on SSD for the `mail-server` workload.

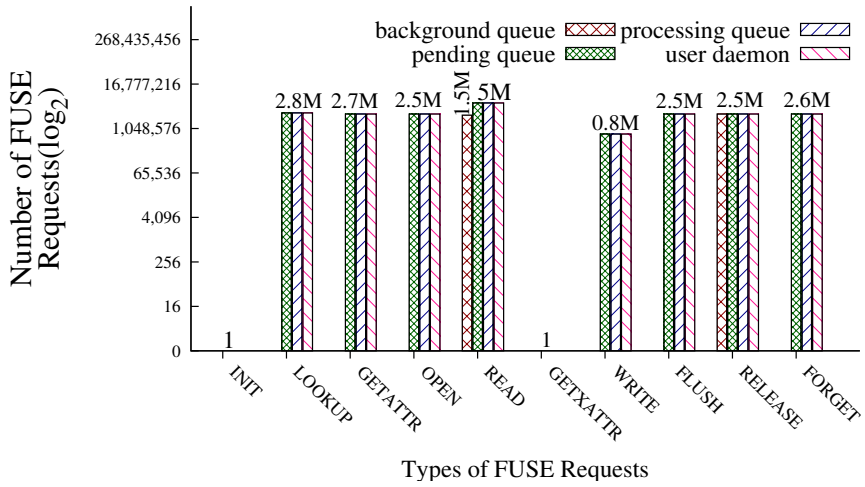


Figure 5.9: Different types of requests that were generated by StackfsBase on SSD for the `web-server` workload.

**Web Server** Figure 5.9 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 Webpages. 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 for the same reasons as for `mail-server` (high speed of SSD).

# Chapter 6

# Related Work

Many researchers used FUSE to implement file systems [8, 16, 27, 56] but little attention was given to understanding FUSE's underlying design and performance. To the best of our knowledge, only two others studied some aspects of FUSE. First, Rajgarhia and Gehani evaluated FUSE performance with Java bindings [42]. 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 [51]. Furthermore, they evaluated only default FUSE configuration and discussed only FUSE's high-level architecture. In this thesis 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 [48]. To improve FUSE performance, Narayan et al. proposed to marry in-kernel stackable file systems [59] with FUSE [38]. Shun et al. modified FUSE's kernel module to allow applications to access storage devices directly [29]. These improvements were in research prototypes and were never included in the mainline.

# Chapter 7

# Conclusions

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%. Finally, in terms of CPU utilization, the relative increase was 31%.

**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.

# Bibliography


[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] Magnus Ahltorp, Love Hornquist-Astrand, and Assar Westerlund. Porting the arla file system to windows NT (2000). In *Proceedings of Management and Administration of Distributed Environments Conference (MADE)*, 2000.

[3] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 77–90, Anaheim, CA, January 1997. USENIX Association.

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

[5] AT&T Bell Laboratories. *Plan 9 – Programmer's Manual*, March 1995.

[6] AVFS: A Virtual Filesystem. *http://avf.sourceforge.net/*.

[7] Jens Axboe. Per backing device write back, 2009. *https://linuxplumbersconf.org/2009/slides/Jens-Axboe-lpc2009-slides-axboe.pdf*.

[8] 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/*.

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

[10] Claudio Calvelli. PerlFS. *http://perlfs.sourceforge.net/*.

[11] 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.

[12] 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.

[13] Mark Carlson, Alan Yoder, Leah Schoeb, Don Deel, Carlos Pratt, Chris Lionetti, and Doug Voigt. SNIA - advanced storage and information technology: Software defined storage, January 2015. *http://datastorageasean.com/sites/default/files/snia_software_defined_storage_white_paper_v1.pdf*.

[14] 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.

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

[16] 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.

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

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

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

[20] Jeremy Fitzhardinge. Userfs. *www.goop.org/~jeremy/userfs/*.

[21] The NetBSD Foundation. The anykernel and rump kernels, 2013. *http://wiki.netbsd.org/rumpkernel/*.

[22] Fuse - bdi max ratio discussion. *https://sourceforge.net/p/fuse/mailman/message/35192764/*.

[23] Fuse - destroy message discussion. *https://sourceforge.net/p/fuse/mailman/message/27787354/*.

[24] Fuse - forgets queue. *https://sourceforge.net/p/fuse/mailman/message/26659508/*.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

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

[33] FUSE for macOS, March 2013. *https://osxfuse.github.io/*.

[34] Pavel Machek. PODFUK—POrtable Dodgy Filesystems in Userland (hacK). *http://atrey.karlin.mff.cuni.cz/~machek/podfuk/podfuk-old.html*.

[35] Pavel Machek. UserVFS. *http://sourceforge.net/projects/uservfs/*.

[36] 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.

[37] Sebastian Messmer. CryFS : Secure encrypted cloud file system, 2015. *https://www.cryfs.org/howitworks*.

[38] 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.

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

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

[41] 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.

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

[43] Nikolaus Rath. S3QL : File system that stores all its data online using storage services like Amazon S3, Google Storage or Open Stack, 2013. *https://github.com/s3ql/s3ql*.

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

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] The linux kernel page cache. *http://sylab-srv.cs.fiu.edu/lib/exe/fetch.php?media=paperclub:lkd3ch16.pdf*.

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

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

[56] 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.

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

[58] 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.

[59] 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.

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

[61] 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.