Performance and Resource Utilization of FUSE User-Space File Systems

BHARATH KUMAR REDDY VANGOOR, Avere Systems, Inc.
PRAFFUL AGARWAL, MANU MATHEW, ARUN RAMACHANDRAN, and SWAMINATHAN SIVARAMAN, Stony Brook University
VASILY TARASOV, IBM Research - Almaden
EREZ ZADOK, Stony Brook University

Traditionally, file systems were implemented as part of operating systems kernels, which provide a limited set of tools and facilities to a programmer. As the 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 article, we analyze the design and implementation of a well-known user-space file system framework, FUSE, for Linux. We characterize its performance and resource utilization for a wide range of workloads. We present FUSE performance and also resource utilization 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 (throughput) caused by FUSE can be completely imperceptible or as high as −83%, even when optimized; and latencies of FUSE file system operations can be increased from none to 4× when compared to Ext4. On the resource utilization side, FUSE can increase relative CPU utilization by up to 31% and underutilize disk bandwidth by as much as −80% compared to Ext4, though for many data-intensive workloads the impact was statistically indistinguishable. Our conclusion is that user-space file systems can indeed be used in production (non-"toy") settings, but their applicability depends on the expected workloads.

CCS Concepts: • **General and reference → Measurement**; **Performance**; • **Software and its engineering → File systems management**; **Software performance**;

Additional Key Words and Phrases: User-space file systems, Linux FUSE

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

Authors' addresses: B. K. R. Vangoor, P. Agarwal, M. Mathew, A. Ramachandran, and S. Sivaraman, 336 New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424, USA; emails: vangoor.bharath@gmail.com, {praagarwal, manmathew}@cs.stonybrook.edu, {arunrm87, sswaminathan92}@gmail.com; V. Tarasov, IBM Research - Almaden, 650 Harry Road, San Jose, California 95120-6099, USA; email: vtarasov@us.ibm.com; E. Zadok, 349 New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424, USA; email: ezk@cs.stonybrook.edu.

1553-3077/2019/05-ART15 $15.00
https://doi.org/10.1145/3310148

ACM Reference format:
Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. 2019. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Trans. Storage* 15, 2, Article 15 (May 2019), 49 pages.
https://doi.org/10.1145/3310148

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, 29], the status quo was always to implement file systems as part of the monolithic OS kernels [12, 37, 51]. Kernel-space implementations avoid the potentially high overheads of passing requests between the kernel and user-space daemon (FUSE daemon)—communications that are inherent to the user-space and micro-kernel implementations [15, 27].

Slowly and over time—although perhaps not overly noticeable—user-space file systems have come back into today's storage systems. In recent years, user-space file systems rose in popularity. Some of the signs and indicators that help prove and support this trend are as follows:

(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 [32, 48]).
(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, 17, 28, 58].
(3) Several existing kernel-level file systems were ported to user space (e.g., ZFS [62], NTFS [42]). Some tried to bring parts of file systems to user space as specialized solutions [49, 60] or port to Windows [2].
(4) Perhaps the greatest endorsement, an increasing number of companies rely on user-space implementations for their storage products: IBM'S GPFS [47] and LTFS [43], Nimble Storage's CASL [41], Apache's HDFS [4], Google File System [26], RedHat's GlusterFS [46], Data Domain's deduplication file system [63], and more. Some implement file systems in user space to store data online in clouds using services such as Google Drive, Amazon S3 [45], and DropBox [38].

Although user-space file systems did not displace kernel-level file systems entirely, and it would be incorrect and premature to assume it at this time, 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 can reach millions (e.g., IBM's GPFS [47]). 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 lightweight rump kernel; these drivers included file systems as well

[40]. A number of frameworks for writing user-space file systems appeared [3, 5, 6, 10, 21, 35, 36]. We now discuss some of the reasons for the rise in the popularity of user-space file systems in recent times:

(1) **Development 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 a 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 the case of distributed file systems, whose clients often run on multiple OSes [60]. For example, 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.
(4) **Existing Code and User Base.** 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 prefer 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 [33, 57, 59]. 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 one indication of these debates [33].

The debates center around two tradeoff factors: (1) how large is the performance overhead caused by 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 studied. Oddly, little has been published on the performance of user-space file system frameworks.

In this article, we use the most common user-space file system framework, *FUSE* for 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) Little information on FUSE's internals is available publicly;
(2) FUSE's source code can be overwhelming to analyze, with complex asynchrony and interactions between user-level and kernel-level components;
(3) 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,
(4) understanding FUSE's design is crucial for the analysis presented in this article.

We developed a simple pass-through stackable file system using FUSE, called *StackFS*, which we layer on top of Ext4. 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. We evaluated StackFS's 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 the native Ext4 file system; but in the worst cases, FUSE can perform 3× slower than the underlying Ext4 file system. For the least friendly workload, `rnd-wr-32th-1f`, FUSE can consume as much as 18× more CPU cycles than Ext4: between 6% and 24%.

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

(1) FUSE Kernel Instrumentation:
https://github.com/sbu-fsl/fuse-kernel-instrumentation.git
(2) FUSE Library Instrumentation:
https://github.com/sbu-fsl/fuse-library-instrumentation.git
(3) Workloads, Results, and Stackable File system implementation:
https://github.com/sbu-fsl/fuse-stackfs.git

Additional information about the code can be found at the aforementioned link, http://www.filesystems.org/fuse/. We then used this instrumentation to identify bottlenecks in FUSE and explain why it performs well for some workloads while poorly for others. For example, we demonstrate that currently, FUSE cannot prefetch or compound [14] small random reads and therefore performs poorly for workloads with many small operations.

The rest of this article is organized as follows. Section 2 discusses the FUSE architecture and implementation details. Section 3 describes our stackable file system's implementation and useful instrumentation that we added to FUSE's kernel and user library. Section 4 introduces our experimental methodology. The bulk of our evaluation and analysis is in Section 5. Then we discuss the related work in the Section 6. We conclude and describe future work in Section 7. In addition to the above mentioned sections, we have an online Appendix [22] which mentions various mount options (arguments) supported by the FUSE library. Also, it describes FUSE's library supported APIs and their arguments in detail.

2 FUSE DESIGN

FUSE—Filesystems in User space—is a well-known user-space file system framework [52]. Its implementation appeared first for Linux-based OSes but over time it was ported to several other OSes [34, 62]. According to modest estimates, at least 100 FUSE-based file systems are readily available on the Web [53]. Although other, specialized implementations of user-space file systems exist [47, 49, 60], we selected FUSE for this study because of its popularity.

Many file systems were implemented using FUSE—thanks mainly to the simple API it provides—yet little work was published on understanding its internal architecture, implementation, and performance [44]. 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 important implementation details. FUSE is available for several OSes: we selected Linux due to its widespread use. We analyzed the code of and ran experiments on the latest stable version of the Linux kernel available at the start of the project—v4.1.13. We opted to use the latest FUSE library commit #386b1b (that was available at the start of the project) rather than the latest versioned release v.2.9.4 because the commit included several important patches (65 files changed, 2,680 insertions, 3,693 deletions) which we did not want excluded from our evaluation. Most of the commits included fixes for various defects encountered during testing.

In the rest of this section, we detail FUSE's design, starting with a high-level architecture in Section 2.1. We describe how the FUSE daemon and the kernel driver communicate in Section 2.2. Several different API levels available in the FUSE user library are discussed in Section 2.3 (e.g., high- vs. low-level APIs). Section 2.4 describes the per-connection session information exchanged through the FUSE user-kernel protocol. Different types of queues, which are part of the FUSE kernel driver, are described in Section 2.5. Section 2.6 describes the FUSE capability of zero-copy using splice and related special buffers that are maintained internally. Section 2.7 discusses FUSE's notification mechanism. Section 2.8 describes how FUSE's library supports multi-threading and processes requests in parallel. In Section 2.9, we discuss important parameters associated with FUSE's write-back in-kernel cache, and how these values affect FUSE. Finally, we discuss various command-line and miscellaneous options that FUSE currently supports in an Appendix [22].

2.1 High-Level Architecture

FUSE consists of a kernel part and a FUSE daemon in the user space. 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 FUSE daemons. File systems of *fuse* type do not require an underlying block device and are usually stackable, in-memory, or network file systems. The *fuseblk* type, however, 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 single-device file systems, however, the *fuseblk* type provides the following features:

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

In addition to these features, *fuseblk* also provides the ability to synchronously unmount the file system: when the file system is the last one 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 behavior 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., 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): they present whole families of file systems. To differentiate between different *mounted* FUSE file systems, the `/proc/mounts` file includes an additional name identifier and 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` character device. This device serves as an interface between user-space FUSE daemons and the kernel. In general, the FUSE daemon reads FUSE requests from `/dev/fuse`, processes them, and then writes the replies back to `/dev/fuse`.

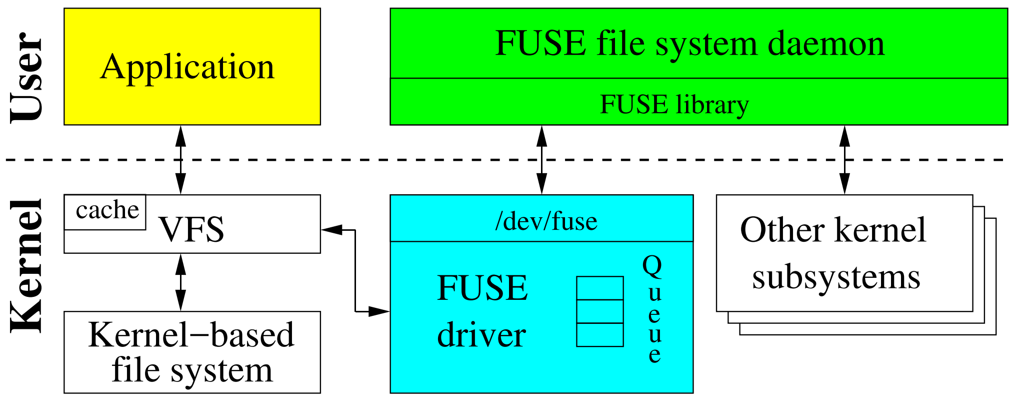

Fig. 1. FUSE high-level architecture.

Figure 1 shows FUSE's high-level architecture. When a user application performs some operation on a mounted FUSE file system, the VFS routes the operation to FUSE's kernel (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. The FUSE 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 the case of a stackable FUSE file system, the FUSE daemon submits operations to the underlying file system (e.g., Ext4); in the case of a block-based FUSE file system, the FUSE daemon reads or writes from the block device; and in the 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 file whose pages are cached in the kernel page cache, do not need to be forwarded to the FUSE driver.

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).

2.2 Kernel-User Protocol

When FUSE's kernel driver communicates with the FUSE daemon, it forms a *FUSE request* structure. The header files `include/uapi/linux/fuse.h` (for use in FUSE's kernel code) and `include/fuse_kernel.h` (for use in FUSE's user-level library) are identical; they define the request 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 FUSE daemon) and output replies (from the FUSE daemon). They also define operation-specific headers: `struct fuse_read_in/fuse_read_out` (for read operations), `struct fuse_fsync_in/out` (for fsync operations), and so forth. Requests have different types (stored in the `opcode` field of the `fuse_in_header`) depending on the operation they encode. The parameter MIN_BUFSIZE (which is equal to 136KB: 132KB for data plus 4KB for headers) is defined in the FUSE library and determines how much data the library requests from the

Table 1. FUSE Request Types Grouped by Semantics

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

The number in parentheses is the size of the corresponding group. Request types that we discuss in the text are typeset in bold.

kernel while reading from the FUSE channel (established during creation of a FUSE mountpoint). However, many FUSE requests are smaller than MIN_BUFSIZE (e.g., 4KB). Therefore, MIN_BUFSIZE defines only the maximum size of the request. When the FUSE library reads the request into the user-space buffer, it first processes the headers (e.g., `struct fuse_in_header` and `struct fuse_write_in` for writes). After determining the opcodes and other request metadata from the headers, the remainder of the request is processed.

Table 1 lists all 43 FUSE request types, grouped by their semantics. As seen, most requests have a direct mapping to traditional VFS operations: we omit discussion of obvious requests (e.g., READ, CREATE) and instead focus next on those less intuitive request types (highlighted in bold in Table 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 (e.g., v7.23, the version we used);
(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 FUSE 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 FUSE daemon to exit gracefully. Currently, a DESTROY request is sent to FUSE daemon only in the case of *fuseblk* but not in the case of the *fuse* file systems types (as mentioned in Section 2.1). Additionally, this DESTROY request is synchronous in nature, blocking the unmount process until it receives a response from the FUSE daemon. The reason for this behavior is because *fuseblk* is mounted with a privileged user credential (access to block device); and in this case, unmount needs to handle flushing of all remaining buffers during the unmount. Unmounting in the case of a *fuse* file system, however, need not wait on the unprivileged FUSE daemon [24].

The INTERRUPT request is generated by the kernel if any requests that were previously requested and passed to the FUSE daemon are no longer needed by the process (e.g., when a user process blocked on a READ request is terminated/cancelled). INTERRUPT requests take precedence over other requests, so the user-space file system will receive queued INTERRUPT requests before any other requests (that are already waiting). The user-space file system may ignore the INTERRUPT requests entirely (when not implemented), or it 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). When removing an inode from the dcache, the kernel passes the FORGET request to the FUSE daemon. FUSE's inode reference count (in user-space file systems) grows by one with every reply to LOOKUP, CREATE, and so forth, requests. FORGET requests pass an `nlookups` parameter which informs the user-space file system (the FUSE daemon) how many lookups to forget. At this point, the user-space file system (the FUSE daemon) might decide to deallocate any corresponding data structures (once their reference count goes to 0). BATCH_FORGET allows the kernel to forget multiple inodes with a single request.

An OPEN request is generated, not surprisingly, when a user application opens a file. 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, and so forth.

OPENDIR and RELEASEDIR requests have the same semantics as OPEN and RELEASE, respectively, but for directories. The READDIRPLUS request returns one or more directory entries, like READDIR, but it also includes metadata information for each entry. This allows the kernel to pre-fill its inode cache, similar to NFSv3's READDIRPLUS procedure [9].

The ACCESS request is generated only in two special cases: `access(2)` and `chdir(2)`. In all other cases, the access checking is done in the actual operation (e.g., an MKDIR request is sent and the fuse daemon can return EACCESS if access is denied). By handling this ACCESS request, the FUSE daemon can implement logic for custom permissions. 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 (i.e., 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 by RENAME2 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 does not exist, then rename fails with `ENOENT` instead of replacing the target. Otherwise, RENAME2 will attempt to swap the two names atomically, as long as the file system is modified only from within the FUSE mount. If no flags are passed, then the kernel falls back to using a regular RENAME request.

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

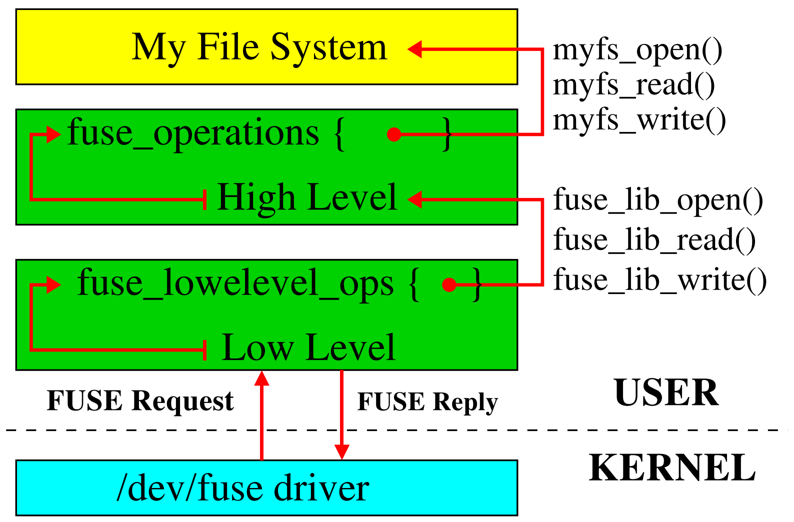

Fig. 2. Interaction of FUSE library levels. "My File System" is an illustrative user-space file system implemented with the high-level API.

FUSE supports file locking using the following request types:

(1) GETLK checks to see if there is already a lock on the file, but does not set one.
(2) SETLKW obtains the requested lock. If the lock cannot be obtained (e.g., someone else owns the lock already), then wait (block) until the lock is released 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 comprises two API levels, as seen in Figure 2. The lower level takes care of the following:

(1) receiving and parsing requests from the kernel;
(2) sending properly formatted replies;
(3) facilitating file system configuration and mounting; and
(4) hiding potential version differences between kernel and user 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, neither inodes nor lookup operations exist in the high-level API, easing the code development. Instead of using inodes, all high-level API methods operate directly on file paths. As a result, FORGET inode methods are also not needed in the high-level API. The high-level API also handles request interrupts and provides other convenient features; for example, developers can use the more common `chown()`, `chmod()`, and `truncate()` methods, instead of the lower-level `setattr()` one. 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 such as `fuse_lib_⋆()` (e.g., `fuse_lib_read`). These functions internally call user-defined high-level API functions, depending on the functionality.

File system developers must decide which API to use, by balancing flexibility vs. development ease. 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 different methods in the `fuse_lowlevel_ops` need to be implemented. The methods share similar names between the two APIs, but differ mainly in parameters and return values. Many methods in both structures are optional. For example, if a file system does not support extended attributes, then the corresponding methods can be left unimplemented. Also, if a FUSE file system does not cache any inodes, then the FORGET method can be left unimplemented.

Many methods in both APIs are similar. We highlight three important differences between the low-level and high-level APIs. First, for more flexibility, low-level methods always take a FUSE request as an argument. Second, method definitions have an even closer correspondence with Linux VFS methods because they often operate on inodes (or rather inode numbers). For example, the VFS lookup method takes a parent inode number to look inside, the name of the file to lookup, and flags to control the lookup mode. It then recursively looks up path components and finally gets the dentry of the file. The lookup method present in the FUSE low-level API takes, in addition to the complete FUSE request, the parent inode number and the name of the file to look up. It performs a similar recursive lookup and gets the attributes of the directory entry. 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 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 part 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 Per Connection Session Information

In Section 2.2, we described how FUSE's user-kernel protocol helps communication between the FUSE kernel and user daemon. FUSE's user-kernel protocol also provides a mechanism to store some in-memory information for each opened file/directory. The ability to store or not store information makes the protocol stateful and stateless, respectively. For each operation in the low-level API (discussed in Section 2.3) that deals with files/directories, there is a `struct fuse_file_info` passed as an argument. This structure is maintained by the FUSE library 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) that can be used to store information about opened files (stateful). When replying to requests like OPEN and CREATE (discussed in Section 2.2), the FUSE daemon can store a 64-bit *file handle* of the opened/created file in this field. The advantage of this is that the `fh` is then passed by the kernel to the FUSE daemon for all operations associated with the opened file. If this is not set in the first place, then the FUSE daemon has to open and close the file for every file operation (statelessness), which could reduce performance. This statefulness applies 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`. 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.

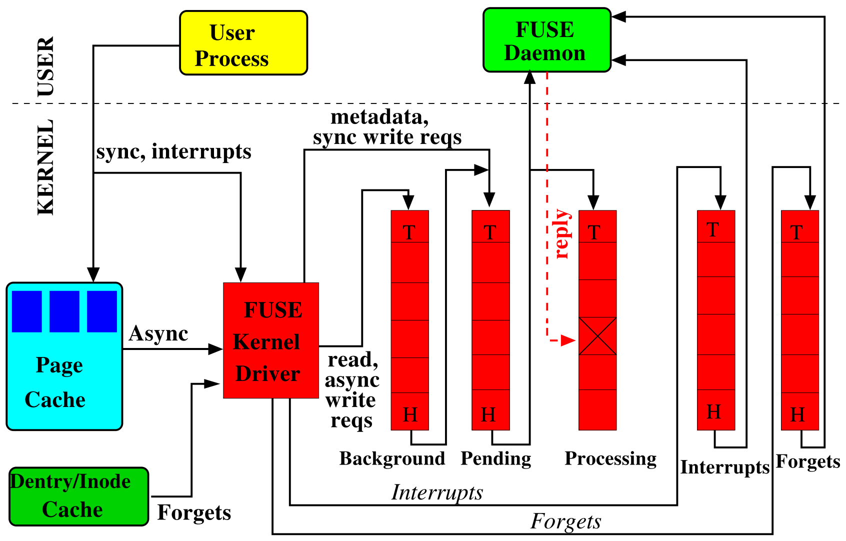

Fig. 3. The organization of FUSE queues marked with their Head and Tail. The processing queue does not have a tail because the FUSE daemon replies in an arbitrary order.

2.5 Queues

In Section 2.1, we described how the FUSE kernel has a request queue for processing by the user space. FUSE actually maintains five different queues as seen in Figure 3: (1) *interrupts* (2) *forgets*, (3) *pending*, (4) *background*, and (5) *processing*. A request belongs to only one queue at any time. FUSE initially puts new INTERRUPT requests in the interrupts queue, FORGET requests in the forgets queue, latency-sensitive requests (e.g., metadata related ones) in the pending queue, and other requests (e.g., data reads or writes) in the background queue.

FUSE uses a separate queue for INTERRUPT requests so they can be processed at a higher priority than other requests. Similarly, FUSE uses a separate queue for FORGET requests to differentiate them from non-forget requests (to balance out the processing of these requests). FORGET requests are sent when the inode is evicted: these requests would be queued 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 observed 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 [25]. Therefore, FUSE maintains a separate queue for FORGET requests and a fair policy is implemented to process them, as explained below.

When a FUSE daemon reads from `/dev/fuse`, requests are transferred to the FUSE daemon as follows:

(1) Priority is given to the requests (if any) in the interrupts queue; they are transferred to the user space before any other request.
(2) FORGET and non-FORGET requests are selected fairly: for each 8 non-FORGET requests in the pending queue, 16 FORGET requests in the forget queue are transferred. This smoothens the burstiness of FORGET requests, while allowing other requests to proceed.

The oldest request in the pending queue is transferred to the user space and simultaneously moved to the processing queue. INTERRUPT and FORGET requests do not have a reply (from the FUSE daemon); therefore, as soon as the FUSE daemon reads these requests, they are terminated.

Thus, *processing* queue requests are currently processed by the FUSE daemon. If the pending queue is empty, then the FUSE daemon is blocked on a read call. When the FUSE daemon replies to a request (by writing 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 requests that are deemed less critical than requests that go directly to the pending queue. For instance, slow data reads and writes typically degrade overall system performance less than slow metadata requests. Therefore, in a default setup (no arguments to FUSE on mount), read requests go to the background queue. Writes also 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 [16]. 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 background 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 FUSE daemon threads invoked in case of the multi-threaded option (discussed in Section 2.8).

The queues' lengths are not explicitly limited. However, when the number of asynchronous requests in the pending and processing queues reaches the value of the tunable `congestion_threshold` parameter (75% of `max_background`, nine 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, to allow the congestion to drain.

2.6 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 [56]. Splicing allows the user space to transfer data between two in-kernel memory buffers without copying the data to user space. This is useful, for example, 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 FUSE 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 each 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. Therefore, each request's header is then read from the pipe using regular `read` calls (i.e., small, at most 80 bytes; memory copying is always performed). 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.

2.7 Notifications

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 FUSE daemon needs to pass some information to the kernel without receiving any previous requests from the kernel. For example, if a user application polls for events on an opened file using the `poll` system call, the FUSE daemon needs to notify the kernel when an event happens that should wake 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 FUSE daemon writes the notification to `/dev/fuse`.

Table 2 describes the six notification types that the FUSE daemon can send to the kernel driver. Usually, FUSE's kernel driver synchronously returns 0 on success or -ERRNO on failure. The only exception is the RETRIEVE notification, for which the kernel replies using a special NOTIFY_REPLY request.

2.8 Multithreading

Initial FUSE implementations supported only a single-threaded daemon but as parallelism got more dominant, both in user applications and hardware, the necessity for 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. In a typical system, there are 10 threads in the thread pool waiting for new requests. After processing a single request, each thread checks to see if there are more than 10 threads running; if so, that thread exits. There is no explicit upper limit on the number of threads created by the FUSE library. An implicit limit exists for two reasons. First, by default, only 12 background requests (`max_background` parameter) can be in the pending queue at one time. Second, the number of foreground requests (the ones that are placed to the pending queue directly) depends on the total amount of I/O activity generated by user processes. However, there is an exception in case of FORGET and BATCH_FORGET requests: no additional thread is created because many such requests are created but they take a small amount of time to complete. Finally, the INTERRUPT request is processed just like other requests in the pending queue, by spawning additional threads to process each INTERRUPT request. But in a typical system where not many INTERRUPTS are generated, the total number of FUSE daemon threads is at most ($max\_background + number\ of\ requests\ in\ pending\ queue$).

2.9 Linux Write-Back Cache and FUSE

To understand how FUSE uses the Linux Page Cache for writes, it is important to understand Linux's Page Cache internals. Below, Section 2.9.1 describes Linux's Page Cache and Section 2.9.2 describes its parameters. Section 2.9.3 describes FUSE's write-back cache code flow. Finally, Section 2.9.4 describes FUSE's write-back mode.

Table 2. FUSE Notification Types, in the Order of Their Opcodes

Notification Type	Description	Arguments
POLL	Notifies that an event happened on a file for which some user application called `poll()`.	Poll handle that the user process will be notified on when an event (I/O) occurred on a file for which a user application called `poll()`.
INVAL_INODE	Invalidates Page/Data cache for a specific inode.	Inode number, offset in the inode, and amount of Page/Data cache to invalidate.
INVAL_ENTRY	Invalidates parent directory attributes and the dentry matching a ⟨*parent-directory*, *file-name*⟩ pair.	Parent inode number, file name, and file name length.
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.	Parent inode number, child inode number, file name, and file name length.
STORE	Store data in the page cache. The STORE and RETRIEVE (below) notification types can be used by FUSE file systems to synchronize their caches with the kernel's.	Inode number, offset into the file to store to, buffer, and flags (e.g., splice, no splice) controlling the copy.
RETRIEVE	Retrieve data from the page cache. The STORE (above) and RETRIEVE notification types can be used by FUSE file systems to synchronize their caches with the kernel's. The data is returned asynchronously in the form of a NOTIFY_REPLY request.	Inode number, number of bytes to retrieve, offset into the file to retrieve from, and pointer to a reply callback function (`void ⋆`).

All notifications take a FUSE channel as extra argument, through which they communicate the notifications. These notification types return 0 on success or -errno on failure.

2.9.1 Linux Page Cache. The Linux kernel implements a cache called page cache [55]. 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 also 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 the cache. In the case of write operations, there are two major strategies:

(1) **write-through:** a write operation will update both in-memory cache and on-disk file. This is useful when re-reading this data at a later time.

Table 3. Parameter Names and Their Abbreviated Names, Their Types, and Default Values (if Applicable)

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	-

We use the short names throughout this article.

(2) **write-back:** a 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*.

FUSE supports both write-through and write-back strategies. In this section, we focus on write-back strategy as the most complex and high-performing. Dirty pages are written to disk by a group of flusher threads. Dirty page write-back occurs under three situations:

(1) 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 the amount of free memory is above the threshold.
(2) When dirty data becomes older than a specified time threshold, ensuring that dirty data does not remain dirty indefinitely. For this, the flusher thread periodically wakes up and writes out old dirty data, thereby ensuring that no dirty pages remain in the page cache indefinitely.
(3) When a user process invokes the `sync` or `fsync` system calls. In this case all dirty data (for `sync`) and all dirty data belonging to files (for `fsync`) is flushed.

Version 2.6.32 Linux kernel maintains multiple flusher threads where each thread flushes dirty pages only to the disk assigned to it, allowing different threads to flush data at different rates to different disks [7]. 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 next, in Section 2.9.2.

2.9.2 Page Cache Parameters. Table 3 summarizes the parameters discussed in detail below.

■ *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 G.B.R. This value can be changed by the user at `/proc/sys/vm/dirty_ratio`. By default this value is set to 20%.

■ *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/⟨bdi⟩/min_ratio` after the mount, where ⟨*bdi*⟩ 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/⟨bdi⟩/max_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:

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

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, 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:

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

■ *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 uses BDI for flushing is responsible for incrementing/decrementing the values for this parameter.

2.9.3 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 actions are performed in order:

A. The page is marked as dirty and `BDI_RECLAIMABLE` is incremented.
B. The `balance_dirty_pages_ratelimited` function is called; it checks whether the task that is generating a dirty page 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 page 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.
C. The `balance_dirty_pages` function is the heart of the write-back cache, as it is the place where write-back and throttling is performed. The following are the main steps being executed as part of this function:
   (1) 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 uses BDIs), BDI dirty pages are calculated as follows:

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

   (2) 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 that this task can dirty before checking this condition again. Thus, it is calculated as follows:

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

   (3) If the above condition fails, then the flusher thread is woken up (if not already in progress) and dirty data is flushed.
   (4) 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 since the end of the previous pause and `task_rate_limit` is the dirty throttle rate, which depends on the write bandwidth of the device. The above calculated pause time is capped from the top and bottom by BDI's `max_pause_time` `min_pause_time` parameters, respectively.
   (5) 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).

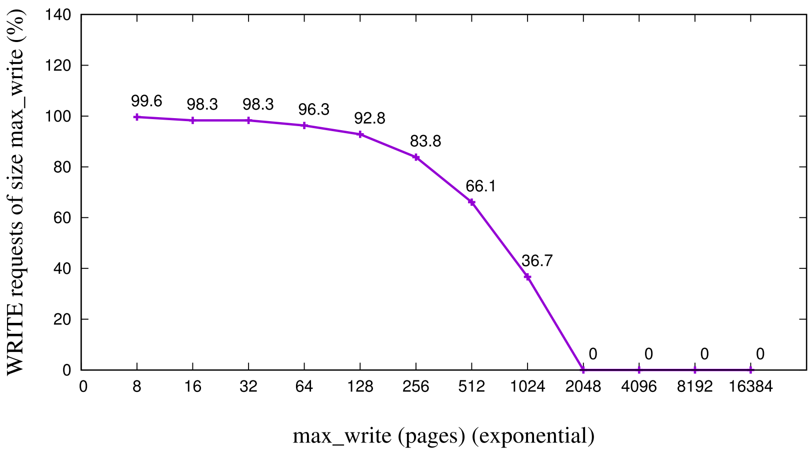

Fig. 4. Percentage of FUSE WRITE requests of `max_write` size depending on different `max_write`'s value.

2.9.4 FUSE Write-Back and Max Write. The basic write behavior of FUSE is write-through and only 4KB of data is sent to the FUSE daemon for writing. This results in performance problems on certain workloads (e.g., when the input I/O size is a multiple of 4KB); when copying a large file into a FUSE file system, `/bin/cp` indirectly causes every 4KB of data to be sent to user space synchronously. The solution FUSE implemented was to use Linux's page cache to support a write-back policy and then make writes asynchronous.

Interestingly, the default B.Mx.R in FUSE is set to only 1%. For example, on a typical Linux system with RAM size equal to 4GB, FUSE's 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. This means that FUSE caches at most 8MB (2,000 pages) of dirty data which can degrade write performance significantly compared to in-kernel file systems that can buffer much more dirty data. The reason that FUSE sets a conservative B.Mx.R limit (1%) is to avoid consuming too much dirty data in case user-level daemon stalls for any reason [23]. The B.Mn.R value, however, can be safely increased if the FUSE 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 on, file data can be pushed to the FUSE daemon in larger chunks of configurable `max_write` size. But write requests still cannot be larger than 8MB (in the example above) because there is at most 8MB of dirty data in the cache. To demonstrate this, we conducted an experiment in which we varied the `max_write` parameter value from 32KB (i.e., 8 pages) to 64MB (i.e., 16,384 pages), in powers of two. (We did a minor change to FUSE's code to increase the maximum size of FUSE's write requests from 128KB to 64MB.) Figure 4 shows the percentage of FUSE write requests that were completely filled with `max_write` amount of data across different `max_write` values when a 60GB file was written using 1MB I/O. Since we used a system with 4GB of RAM for this experiment, the maximum page cache size available to FUSE file systems was 8MB. Therefore, as `max_write` approaches 8MB the percentage of full requests (a request is full if its size is `max_write`) drops. And none of the FUSE write requests are full when the `max_write` value is above 8MB (i.e., 2,048 pages).

3 IMPLEMENTATIONS

To study FUSE's performance, we developed a simple stackable pass-through file system, called *Stackfs*. We also 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.

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 unmodified FUSE requests 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.

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 look up 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 show the key parts of the structure for Stackfs's inode. This structure is 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—if it is not found in a local hash table—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). This also causes inode numbers to change from one mount to another, and even within a single mount if an inode was deallocated and then looked up again. 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: if found, Stackfs 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's 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 that 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 store 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 directories similar to 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; importantly, this handle can be stored as part of the `struct fuse_file_info`, which stores information about open files/directories. Below we show 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 is 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

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

3.2 Performance Statistics and Traces

The existing FUSE instrumentation in Linux was insufficient for in-depth FUSE performance analysis. We therefore instrumented FUSE to collect and provide 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 range of 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 of efficiently recording a log_2 latency histogram was introduced first in OSprof [31].)

We then added four such arrays to FUSE: the first three arrays are in the kernel (in `fuse_conn` structure) and are assigned to each of FUSE's three main queues: background, pending, and processing. We did not instrument the interrupt queue, because we were mainly interested in the file system operations. We also did not instrument the forget queue, as we never used the BATCH-FORGET optimization, and the forget queue is only meant for a delete operation; the fourth array is in user space (in `fuse_session` structure) and tracks the time the FUSE 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 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 FUSE daemon. When triggered, the handler prints the array to a log file specified during the FUSE daemon's start. 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 FUSE daemon runs in debug mode but there is no tracing support for FUSE's kernel module. We used Linux's static trace point mechanism [18] to add over 30 trace points to FUSE's kernel module. These new trace points are mainly to monitor the creation of requests during the complex write-back logic, track the amount of data being read/written per request, and track metadata operations (to know how often they get generated). These trace points were not enabled during the performance experiments, so as not to incur unnecessary overheads. Figure 5 shows different trace points that we tracked during a normal FUSE request flow.

The trace points, in order, correspond to the following operations (using write request as an example):

(1) User request being passed to the VFS layer.
(2) After the VFS layer processes the request, it passes it to the FUSE kernel.
(3) After FUSE request allocation and initialization is complete, the FUSE request is passed to the appropriate FUSE kernel queue (pending queue here).
(4) The FUSE request is copied to the user-space daemon.
(5) The FUSE daemon processes the request and passes it to the Ext4 layer.
(6) The Ext4 layer, after execution, passes the reply to the FUSE daemon.
(7) The FUSE daemon processes the reply from the Ext4 layer.
(8) The FUSE daemon sends the reply to the FUSE request in the pending queue in the FUSE kernel. The FUSE kernel processes the reply.
(9) The FUSE kernel sends the reply to the VFS layer, which processes it.
(10) The VFS layer replies to the user request.

Table 4 details the above trace points, with additional information about the average latencies of a write request generated by Stackfs for a `seq-wr-4KB-1th-1f` workload (explained in detail in Section 4.2) with HDD, across the multiple profile points.

Tracing helped us learn how fast 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

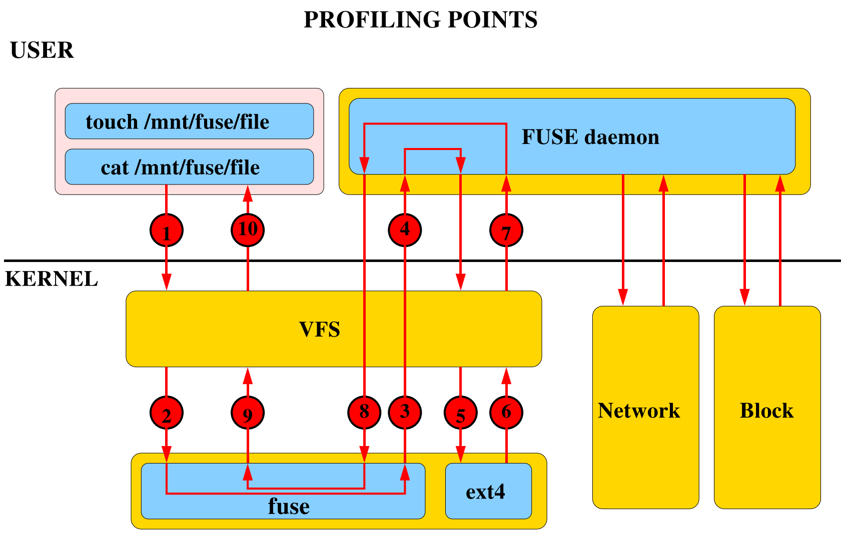

Fig. 5. Locations of trace points throughout the flow of FUSE requests. The circled numbers represent the flow of operations throughout the FUSE system.

Table 4. Average Latencies (CPU Time) of a Single Write Request Generated by Stackfs During Sequential Write by One Thread in 4KB Units on HDD, Across Multiple Profile Points

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

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 the `master` GIT branch of the following repositories:

(1) FUSE Kernel Instrumentation: https://github.com/sbu-fsl/fuse-kernel-instrumentation.git
(2) FUSE Library Instrumentation: https://github.com/sbu-fsl/fuse-library-instrumentation.git

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 section, we detail our evaluation 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. Following this chapter, Chapter 5 presents the actual evaluation results.

4.1 Configurations

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

(1) A *basic* configuration (called *SBase*) with no major FUSE optimizations.
(2) An *optimized* configuration (called *SOpt*) that enables all FUSE improvements available as of this writing.

Compared to SBase, the SOpt 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 FUSE daemon runs in multi-threaded mode; and
(4) splicing is activated for all operations to reduce user/kernel data copies (using the `splice_read`, `splice_write`, and `splice_move` parameters).

We left all other parameters at their default values in both configurations: for example, `max_readahead`, which denotes the maximum amount of data that may be cached in the kernel preemptively based on expected reads, defaults to 32 pages, and `async_read`, which enables support for asynchronous read requests, is on.

We evaluated the impact of SBase and SOpt (with all the optimizations) on all workloads. Apart from that, in order to assess the impact of the individual optimizations, we experimented and assessed the impact of each incremental combination of optimizations on a subset of workloads. The following optimizations were evaluated individually:

— splice,
— big_writes and/or max_write,
— writeback_cache, and
— multi-threading.

All the parameters except multi-threading are described in an Appendix [22].

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 5 describes all the workloads that we employed in the evaluation. Generally, micro-workloads focus on specific operations whereas macro-workloads mix all sorts of operations to emulate more realistic user/system "real world" workloads; data-intensive micro-workloads focus on reads and writes, whereas metadata-intensive micro-workloads focus on metadata (and namespace) operations such as file creations and deletions. 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 are sufficient to fully exercise all CPU cores in our system. We selected dataset sizes so that for all workloads they are larger than the RAM available on the

Table 5. Description of Workloads

Workload Name	Workload Type	Description
`rnd-rd-Nth-1f`	micro, data	*N* threads [1, 32] randomly read from a single preallocated 60GB file
`rnd-wr-Nth-1f`	micro, data	*N* threads [1, 32] randomly write to a single preallocated 60GB file
`seq-rd-Nth-1f`	micro, data	*N* threads [1, 32] sequentially read from a single preallocated 60GB file
`seq-wr-1th-1f`	micro, data	A single thread creates and sequentially writes a new 60GB file
`seq-rd-32th-32f`	micro, data	32 threads sequentially read 32 preallocated 2GB files, where each thread reads its own file
`seq-wr-32th-32f`	micro, data	32 threads sequentially write 32 new 2GB files, where each thread writes its own file
`files-cr-Nth`	micro, metadata	*N* threads [1, 32] create 4 million 4KB files over 1,000 directories
`files-rd-Nth`	micro, data, metadata	*N* threads [1, 32] read from 1 million preallocated 4KB files over 1,000 directories
`files-del-Nth`	micro, metadata	*N* threads [1, 32] delete 4 million of preallocated 4KB files over 1,000 directories
`web-server`	macro	Web-server workload emulated by Filebench, scaled up to 1.25 million files
`mail-server`	macro	Mail-server workload emulated by Filebench, scaled up to 1.5 million files
`file-server`	macro	File-server workload emulated by Filebench, scaled up to 200,000 files

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

test node. In this article, 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 1.5-alpha3 [20, 54] 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;
—Branch: `master`.

4.3 Experimental Setup

FUSE's performance impact depends heavily on the speed of the underlying storage: faster devices expose FUSE's own overheads. We therefore experimented with two common storage devices of different speeds: an HDD (Seagate Savvio 15K.2, 15KRPM, 146GB) and an SSD (Intel X25-M SSD, 200GB) (We leave RAID and ramdisk workloads to future work.) 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.

We used Ext4 [19] as the underlying file system because it is common, stable, and has a well-documented design that facilitates performance analysis. Before every experiment, we reformatted the storage devices with Ext4 and remounted the file systems. We observed high standard deviation in our experiments with Ext4's default lazy inode initialization mode. To lower the variability in our experiments, we disabled Ext4's lazy inode initialization [11].

We measured the following four parameters—throughput, latency, CPU utilization, and disk bandwidth—and calculated the standard deviations for each parameter. After all runs, the measured standard deviations in our experiments were less than 2% for all workloads except for three workloads: `rnd-wr-1th-1f` (6%), `files-rd-32th` (7%), and `mail-server` (6%)—we feel these three are still small enough that they do not merit a deeper investigation.

5 EVALUATION

For many, FUSE is just a practical *tool* to build real products or prototypes, but not a research focus. This article represents a large number of experiments and many details are presented; and we recognize that different readers may desire different levels of verbosity. Therefore, to present our results more effectively, we describe the evaluation in three sections corresponding to different levels of detail. First, Section 5.1 presents the all-inclusive results tables but provides only an overview of the results without explaining their meaning yet—the most useful information for many practitioners. Next, Section 5.2 provides the detailed analysis of our extensive evaluation results. Finally, for those who wish to jump right into a quick summary of our conclusions, Section 5.3 provides a concise list of our conclusions and observations.

5.1 Overview of Results

5.1.1 Performance Overview. To evaluate the performance degradation caused by FUSE, we measured the throughput (in ops/sec) and the latency (in ms/op) achieved by native Ext4 and then measured the same for Stackfs deployed over Ext4. As detailed in Section 4, we used two configurations of Stackfs: a basic (*SBase*) and an optimized (*SOpt*) one. From here on, we use the term *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 6 shows absolute throughputs for Ext4 and relative performance for the two Stackfs configurations for both HDD and SSD. Table 7 shows the absolute latencies and relative performance, again for the two Stackfs configurations for both HDD and SSD.

Note that in case of Table 6 (throughput observations), a performance improvement is signified by an increase in throughput, so a positive percentage value in the *Stackfs* columns indicates that FUSE is performing better, and a negative value means degradation (lower is worse). For Table 7 (latency observations), it is the opposite: a performance improvement is signified by a decrease in latency, so a negative percentage value in the *Stackfs* columns indicates that FUSE is performing better, and a positive value indicates degradation (higher is worse).

For better clarity, we categorized Stackfs's performance results (from Tables 6 and 7) 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.

Table 6. List of Workloads and Corresponding Throughput Results

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			Ext4 (ops/s)	SBase (%Diff)	SOpt (%Diff)	Ext4 (ops/s)	SBase (%Diff)	SOpt (%Diff)
1	seq-rd-1th-1f	4	38,382	$-2.4^{\dagger}$	$+1.7^{\ddagger}$	30,694	$-0.5^{\dagger}$	$-0.9^{\dagger}$
2		32	4805	$-0.2^{\dagger}$	$-2.2^{\dagger}$	3,811	$+0.8^{\ddagger}$	$+0.3^{\ddagger}$
3		128	1199	$-0.9^{\dagger}$	$-2.1^{\dagger}$	950	$+0.4^{\ddagger}$	$+1.7^{\ddagger}$
4		1024	150	$-0.9^{\dagger}$	$-2.2^{\dagger}$	119	$+0.2^{\ddagger}$	$-0.3^{\dagger}$
5	seq-rd-32th-1f	4	1,228,400	$-2.4^{\dagger}$	$-3.0^{\dagger}$	973,450	$0.0^{\ddagger}$	$+2.1^{\ddagger}$
6		32	153,480	$-2.4^{\dagger}$	$-4.1^{\dagger}$	121,410	$+0.7^{\ddagger}$	$+2.2^{\ddagger}$
7		128	38,443	$-2.6^{\dagger}$	$-4.4^{\dagger}$	30,338	$+1.5^{\ddagger}$	$+2.0^{\ddagger}$
8		1,024	4805	$-2.5^{\dagger}$	$-4.0^{\dagger}$	3,814.50	$-0.1^{\dagger}$	$-0.4^{\dagger}$
9	seq-rd-32th-32f	4	11,141	$-36.9^{\#}$	$-26.9^{\#}$	32,855	$-0.1^{\dagger}$	$-0.2^{\dagger}$
10		32	1491	$-41.5^{\#}$	$-30.3^{\#}$	4,202	$-0.1^{\dagger}$	$-1.8^{\dagger}$
11		128	371	$-41.3^{\#}$	$-29.8^{\#}$	1,051	$-0.1^{\dagger}$	$-0.2^{\dagger}$
12		1,024	46	$-41.0^{\#}$	$-28.3^{\#}$	131	$0.0^{\ddagger}$	$-2.1^{\dagger}$
13	rnd-rd-1th-1f	4	243	-10.0^{*}	-10.0^{*}	4,712	$-32.1^{\#}$	$-39.8^{\#}$
14		32	232	-7.4^{*}	-7.5^{*}	2,032	-18.8^{*}	$-25.2^{\#}$
15		128	191	-7.4^{*}	-5.5^{*}	852	-14.7^{*}	-12.4^{*}
16		1,024	88	-9.0^{*}	$-3.1^{\dagger}$	114	-15.3^{*}	$-1.5^{\dagger}$
17	rnd-rd-32th-1f	4	572	$-60.4^{!}$	-23.2^{*}	24,998	$-82.5^{!}$	$-27.6^{\#}$
18		32	504	$-56.2^{!}$	-17.2^{*}	4,273	$-55.7^{!}$	$-1.9^{\dagger}$
19		128	278	$-34.4^{\#}$	-11.4^{*}	1,123	$-29.1^{\#}$	$-2.6^{\dagger}$
20		1,024	41	$-37.0^{\#}$	-15.0^{*}	126	-12.2^{*}	$-1.9^{\dagger}$
21	seq-wr-1th-1f	4	36,919	$-26.2^{\#}$	$-0.1^{\dagger}$	32,959	-9.0^{*}	$+0.1^{\ddagger}$
22		32	4,615	-17.8^{*}	$-0.2^{\dagger}$	4,119	$-2.5^{\dagger}$	$+0.1^{\ddagger}$
23		128	1,153	-16.6^{*}	$-0.2^{\dagger}$	1,030	$-2.1^{\dagger}$	$+0.1^{\ddagger}$
24		1,024	144	-17.7^{*}	$-0.3^{\dagger}$	129	$-2.3^{\dagger}$	$-0.1^{\dagger}$
25	seq-wr-32th-32f	4	34,370	$-2.5^{\dagger}$	$+0.1^{\ddagger}$	32,921	$0.0^{\ddagger}$	$+0.2^{\dagger}$
26		32	4,296	$-2.7^{\dagger}$	$0.0^{\ddagger}$	4,115	$+0.1^{\ddagger}$	$+0.1^{\ddagger}$
27		128	1,075	$-2.6^{\dagger}$	$0.0^{\ddagger}$	1,029	$0.0^{\ddagger}$	$+0.2^{\ddagger}$
28		1,024	134	$-2.4^{\dagger}$	$-0.2^{\dagger}$	129	$-0.1^{\dagger}$	$+0.2^{\ddagger}$
29	rnd-wr-1th-1f	4	1,074	$-0.7^{\dagger}$	$-1.3^{\dagger}$	16,066	$+0.9^{\ddagger}$	$-27.0^{\#}$
30		32	708	$-0.1^{\dagger}$	$-1.3^{\dagger}$	4,102	$-2.2^{\dagger}$	-13.0^{*}
31		128	359	$-0.1^{\dagger}$	$-1.3^{\dagger}$	1,045	$-1.7^{\dagger}$	$-0.7^{\dagger}$
32		1,024	79	$0.0^{\ddagger}$	$-0.8^{\dagger}$	129	$0.0^{\ddagger}$	$-0.3^{\dagger}$
33	rnd-wr-32th-1f	4	1,073	$-0.9^{\dagger}$	$-1.8^{\dagger}$	16,213	$-0.7^{\dagger}$	$-26.6^{\#}$
34		32	705	$+0.1^{\ddagger}$	$-0.7^{\dagger}$	4,103	$-2.2^{\dagger}$	-13.0^{*}
35		128	358	$+0.3^{\ddagger}$	$-1.1^{\dagger}$	1,031	$-0.1^{\dagger}$	$0.0^{\ddagger}$
36		1,024	79	$+0.1^{\ddagger}$	$-0.3^{\dagger}$	128	$+0.9^{\ddagger}$	$-0.3^{\dagger}$
37	files-cr-1th	4	30,211	$-57.0^{!}$	$-81.0^{!}$	35,361	$-62.2^{!}$	$-83.3^{!}$
38	files-cr-32th	4	36,590	$-50.2^{!}$	$-54.9^{!}$	46,688	$-57.6^{!}$	$-62.6^{!}$
39	files-rd-1th	4	645	$0.0^{\ddagger}$	-10.6^{*}	8,055	-25.0^{*}	$-60.3^{!}$
40	files-rd-32th	4	1,263	$-50.5^{!}$	$-4.5^{\dagger}$	25,341	$-74.1^{!}$	$-33.0^{\#}$

(Continued)

Table 6. Continued

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			Ext4 (ops/s)	SBase (%Diff)	SOpt (%Diff)	Ext4 (ops/s)	SBase (%Diff)	SOpt (%Diff)
41	files-del-1th	-	1,105	$-4.0^{\dagger}$	-10.2^{*}	7,391	$-31.6^{\#}$	$-60.7^{!}$
42	files-del-32th	-	1,109	$-2.8^{\dagger}$	-6.9^{*}	8,563	$-42.9^{\#}$	$-52.6^{!}$
43	file-server	-	1,705	$-26.3^{\#}$	$-1.4^{\dagger}$	5,201	$-41.2^{\#}$	$-1.5^{\dagger}$
44	mail-server	-	1,547	$-45.0^{\#}$	$-4.6^{\dagger}$	11,806	$-70.5^{!}$	$-32.5^{\#}$
45	web-server	-	1,704	$-51.8^{!}$	$+6.2^{\ddagger}$	19,437	$-72.9^{!}$	-17.3^{*}

(4) The *Orange* class ($^{\#}$) indicates that the performance degradation is between [25 and 50)%.
(5) And finally, the *Red* class ($^{!}$) is when performance decreased by more than 50%.

Although the ranges for acceptable performance degradation depend on the specific deployment and the value of other benefits provided by FUSE, our classification gives a broad overview of FUSE's performance. Below we list the main observations that characterize the results. We start with general trends and proceed to more specific results toward the end of the list. We also tie our observations with the relevant row number(s) in Tables 6 and 7.

■ Observation 1. The relative performance difference varied across workloads, devices, and FUSE configurations. For throughput, it varied from −83.1% for `files-cr-1th` [Table 6, row #37] to +6.2% for `web-server` [Table 6, row #45]. For latency, it varied from +604.0% for `files-cr-1th` [Table 7, row #37] to −3.8% for `seq-rd-1th-1f` [Table 7, row #3].

■ Observation 2. For many workloads, FUSE's optimizations improve performance significantly. For example, in the `web-server` workload, SOpt improves throughput performance by 6.2% while SBase degrades it by more than 50% [Table 6, row #45]. Similarly, for the same `web-server` workload, compared to Ext4, latency for SBase increased by over 110.6%, while for SOpt, latency actually decreased by ~1% [Table 7, row #45].

■ Observation 3. Although optimizations improve the performance of some workloads, they can degrade the performance for other workloads. For example, SOpt decreases throughput performance by 35% more than SBase for the `files-rd-1th` workload on SSD [Table 6, row #39], and latency increased for SOpt by ~350% more than SBase for the `files-cr-1th` workload on HDD [Table 7, row #37].

■ Observation 4. For throughput results, in the best-performing configuration of Stackfs (among SOpt and SBase), out of a total 45 workloads, only two file-create workloads fell into the red class: `files-cr-1th` and `files-cr-32th` [Table 6, rows #37–38].

In case of latency results, the same two file-create workloads fell into the red class for HDDs: `files-cr-1th` [Table 7, row #37] and `files-cr-32th` [Table 7, row #38]. But for SSDs, it was four workloads: the same two workloads as for HDD, plus `rnd-rd-1th-1f` (4KB I/O Size) [Table 7, rows #13–16] and `files-del-32th` [Table 7, row #42].

■ Observation 5. Stackfs's performance depends significantly on the underlying device. For example, for sequential read workloads [Table 6, rows #1–12; Table 7, rows #1–12], Stackfs shows no performance degradation for SSD and an appreciable degradation for HDD (26–42% throughput degradation and 14–72% latency degradation). The situation is reversed when a `mail-server` workload is used [Table 6, row #44; Table 7, row #44].

Table 7. List of Workloads and Corresponding Latency Results

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			Ext4 (ms/op)	SBase (%Diff)	SOpt (%Diff)	Ext4 (ms/op)	SBase (%Diff)	SOpt (%Diff)
1	seq-rd-1th-1f	4	0.03	$0.0^{\ddagger}$	$0.0^{\ddagger}$	0.03	$0.0^{\ddagger}$	$0.0^{\ddagger}$
2		32	0.22	$+0.5^{\dagger}$	$+0.5^{\dagger}$	0.26	$0.0^{\ddagger}$	$+0.4^{\dagger}$
3		128	0.86	$-0.1^{\ddagger}$	$+0.1^{\dagger}$	1.04	$-0.1^{\ddagger}$	$-3.8^{\ddagger}$
4		1,024	6.92	$0.0^{\ddagger}$	$0.0^{\ddagger}$	8.31	$+0.4^{\dagger}$	$-3.7^{\ddagger}$
5	seq-rd-32th-1f	4	0.03	$0.0^{\ddagger}$	$0.0^{\ddagger}$	0.03	$0.0^{\ddagger}$	$0.0^{\ddagger}$
6		32	0.22	$0.0^{\ddagger}$	$0.0^{\ddagger}$	0.26	$+0.4^{\dagger}$	$-0.8^{\ddagger}$
7		128	0.87	$-0.1^{\ddagger}$	$-0.6^{\ddagger}$	1.04	$+0.4^{\dagger}$	$-0.4^{\ddagger}$
8		1,024	6.92	$+0.1^{\dagger}$	$+0.1^{\dagger}$	8.33	$0.0^{\ddagger}$	$-0.6^{\ddagger}$
9	seq-rd-32th-32f	4	2.91	$+56.2^{!}$	$+33.7^{\#}$	0.97	$-0.1^{\ddagger}$	$+0.1^{\dagger}$
10		32	21.44	$+72.8^{!}$	$+17.6^{*}$	7.58	$-0.1^{\ddagger}$	$+0.1^{\dagger}$
11		128	86.14	$+71.9^{!}$	$+14.6^{*}$	30.27	$+0.1^{\dagger}$	$+0.1^{\dagger}$
12		1,024	688.71	$+72.1^{!}$	$+17.3^{*}$	242.38	$-0.1^{\ddagger}$	$-0.1^{\ddagger}$
13	rnd-rd-1th-1f	4	4.11	$+10.0^{*}$	$+10.2^{*}$	0.21	$+61.6^{!}$	$+59.7^{!}$
14		32	4.30	$+7.5^{*}$	$+7.7^{*}$	0.49	$+26.9^{\#}$	$+31.6^{\#}$
15		128	5.24	$+7.5^{*}$	$+7.7^{*}$	1.16	$+18.9^{*}$	$+24.3^{*}$
16		1,024	11.60	$+10.0^{*}$	$+3.6^{\dagger}$	8.75	$+17.7^{*}$	$+2.2^{\dagger}$
17	rnd-rd-32th-1f	4	54.62	$+139.8^{!}$	$+23.5^{*}$	1.28	$+478.4^{!}$	$+37.8^{\#}$
18		32	61.72	$+126.7^{!}$	$+20.7^{*}$	7.47	$+124.1^{!}$	$+2.7^{\dagger}$
19		128	115.39	$+49.9^{\#}$	$+13.4^{*}$	28.30	$+40.9^{\#}$	$+2.5^{\dagger}$
20		1,024	777.16	$+59.4^{!}$	$+20.6^{*}$	252.66	$+12.9^{*}$	$+2.5^{\dagger}$
21	seq-wr-1th-1f	4	0.03	$+42.3^{\#}$	$0.0^{\ddagger}$	0.03	$+13.3^{*}$	$0.0^{\ddagger}$
22		32	0.21	$+27.6^{\#}$	$0.0^{\ddagger}$	0.24	$+5.8^{*}$	$0.0^{\ddagger}$
23		128	0.86	$+27.4^{\#}$	$+0.2^{\dagger}$	0.96	$+5.4^{*}$	$-0.1^{\ddagger}$
24		1,024	6.87	$+26.3^{\#}$	$+0.2^{\dagger}$	7.71	$+5.1^{*}$	$-0.1^{\ddagger}$
25	seq-wr-32th-32f	4	0.92	$+2.2^{\dagger}$	$-0.7^{\ddagger}$	0.96	$+0.3^{\dagger}$	$-0.8^{\ddagger}$
26		32	7.35	$+2.1^{\dagger}$	$-0.7^{\ddagger}$	7.69	$+0.4^{\dagger}$	$-1.0^{\ddagger}$
27		128	29.36	$+2.0^{\dagger}$	$-0.6^{\ddagger}$	30.76	$+0.4^{\dagger}$	$-0.8^{\ddagger}$
28		1,024	234.93	$+2.1^{\dagger}$	$-0.4^{\ddagger}$	245.83	$+0.4^{\dagger}$	$-0.7^{\ddagger}$
29	rnd-wr-1th-1f	4	0.94	$+0.4^{\dagger}$	$+1.0^{\dagger}$	0.06	$0.0^{\ddagger}$	$+31.1^{\#}$
30		32	1.43	$+0.1^{\dagger}$	$+0.4^{\dagger}$	0.24	$0.0^{\ddagger}$	$+13.2^{*}$
31		128	2.81	$+0.3^{\dagger}$	$+1.4^{\dagger}$	0.95	$+0.1^{\dagger}$	$+0.5^{\dagger}$
32		1,024	12.55	$+0.5^{\dagger}$	$+1.3^{\dagger}$	7.64	$+0.3^{\dagger}$	$+1.6^{\dagger}$
33	rnd-wr-32th-1f	4	24.76	$+7.9^{*}$	$+14.1^{*}$	1.94	$-0.8^{\ddagger}$	$+35.9^{\#}$
34		32	39.95	$+6.9^{*}$	$+12.5^{*}$	6.88	$+8.6^{*}$	$+24.8^{*}$
35		128	71.37	$+18.3^{*}$	$+23.6^{*}$	25.09	$+16.3^{*}$	$+19.9^{*}$
36		1,024	254.59	$+39.8^{\#}$	$+51.3^{!}$	155.26	$+41.4^{\#}$	$+52.5^{!}$
37	files-cr-1th	4	0.03	$+143.3^{!}$	$+496.7^{!}$	0.03	$+176.0^{!}$	$+604.0^{!}$
38	files-cr-32th	4	0.83	$+114.9^{!}$	$+131.3^{!}$	0.67	$+150.3^{!}$	$+169.4^{!}$
39	files-rd-1th	4	1.52	$+8.0^{*}$	$+13.7^{*}$	0.12	$+35.0^{\#}$	$+156.4^{!}$
40	files-rd-32th	4	24.21	$+109.2^{!}$	$+17.0^{*}$	1.22	$+276.0^{!}$	$+44.8^{\#}$

(Continued)

Table 7. Continued

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			Ext4 (ms/op)	SBase (%Diff)	SOpt (%Diff)	Ext4 (ms/op)	SBase (%Diff)	SOpt (%Diff)
41	files-del-1th	-	0.88	+3.1†	+13.2*	0.13	+47.6$^{\#}$	+165.9$^{!}$
42	files-del-32th	-	28.80	+2.3†	+7.5*	3.65	+71.6$^{!}$	+120.1$^{!}$
43	file-server	-	29.39	+33.3$^{\#}$	+3.1†	9.51	+66.7$^{!}$	+1.3†
44	mail-server	-	10.96	+62.6$^{!}$	+8.6*	1.29	+252.9$^{!}$	+43.4$^{\#}$
45	web-server	-	53.93	+110.6$^{!}$	−1.0‡	4.69	+281.1$^{!}$	+13.9*

■ Observation 6. In the case of throughput results, at least in one Stackfs configuration, all write workloads (sequential and random) [Table 6, rows #21–36] are within the *Green* and *Blue* classes for both HDD and SSD. In the case of latency results for the same write workloads [Table 7, rows #21–36], this holds true for all but four write workloads. The exceptions are the four `rnd-wr-32th` workloads [Table 7, rows #33–36].

■ Observation 7. The performance of sequential read [Table 6, rows #1–12; Table 7, rows #1–12] are well within the *Green*, *Blue* class for both HDD and SSD; however, for the `seq-rd-32th-32f` workload [Table 6, rows #5–8; Table 7, rows #5–8] on HDD, they are in *Orange* class. Random read workload results [Table 6, rows #13–20; Table 7, rows #13–20] span all four classes. Furthermore, the performance grows as I/O sizes increase for both HDD and SSD.

■ Observation 8. In general, Stackfs performs visibly worse for metadata-intensive and macro workloads [Table 6, rows #37–45; Table 7, rows #37–45] than for data-intensive workloads [Table 6, rows #1–36; Table 7, rows #1–36]. The performance is especially low for SSDs.

5.1.2 Resource Utilization Overview. In this section, we evaluate how FUSE utilizes two major computational resources: CPU and disk bandwidth. It is expected that FUSE-based file systems would generally require more CPU cycles than in-kernel file sytems. Additional cycles are spent, for example, on FUSE requests management, memory copying, and context switches. As a result, available disk bandwidth is expected to be less utilized by user-space file systems than by their kernel counterparts. In this section, we investigate, confirm, and quantify these expectations.

CPU Utilization Observations. For a given workload, Filebench repeats the same set of operations in a loop (e.g., the loop of reads in the `seq-wr-1th-1f` workload). A single loop typically represents a logical unit of work in a user application (e.g., servicing a single HTTP request by a Web-server). It is therefore reasonable to measure how much resources a single loop consumes. To evaluate FUSE's resource utilization, we first measured the CPU utilization for each Filebench loop of operations (in CPU nanosec/loop) achieved by native Ext4; we then measured the same for Stackfs deployed over Ext4. We then calculated the relative CPU utilization (or improvement) of Stackfs vs. Ext4 for each workload. As detailed in Section 4, we used two configurations of Stackfs: a basic (*SBase*) and optimized (*SOpt*) one. Again, we use *Stackfs* to refer to both of these configurations.

Table 8 shows the results for the CPU utilization in nanosec/loop for Ext4 and the relative performance for two Stackfs configurations for both the HDD and SSD. As done earlier, we categorized the results (from Table 8) by Stackfs's CPU utilization (relative to Ext4) into five classes. The CPU utilization of Ext4 is denoted by x in all cases below:

(1) The *Blue* class (marked with ‡) indicates that the difference in CPU utilization is ≤ 0. In other words, it indicates a lower or equal CPU utilization by Stackfs compared to native Ext4.

Table 8. List of Workloads and Corresponding CPU-Utilization Results

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			Ext4 (ns/loop)	SBase (factor)	SOpt (factor)	Ext4 (ns/loop)	SBase (factor)	SOpt (factor)
1	seq-rd-1th-1f	4	8.6	$2.2\times^{*}$	$2.5\times^{*}$	7.0	$2.2\times^{*}$	$2.5\times^{*}$
2		32	57.2	$2.4\times^{*}$	$2.8\times^{*}$	46.4	$2.4\times^{*}$	$2.9\times^{*}$
3		128	221.1	$2.4\times^{*}$	$2.8\times^{*}$	179.4	$2.4\times^{*}$	$2.9\times^{*}$
4		1,024	1,692.4	$2.5\times^{*}$	$2.8\times^{*}$	1,379.1	$2.4\times^{*}$	$2.9\times^{*}$
5	seq-rd-32th-1f	4	7.4	$1.1\times^{\dagger}$	$1.04\times^{\dagger}$	6.0	$1.1\times^{\dagger}$	$1.1\times^{\dagger}$
6		32	29.9	$1.1\times^{\dagger}$	$1.2\times^{\dagger}$	24.3	$1.1\times^{\dagger}$	$1.2\times^{\dagger}$
7		128	118.6	$1.1\times^{\dagger}$	$1.2\times^{+}$	94.0	$1.2\times^{\dagger}$	$1.2\times^{\dagger}$
8		1,024	1,493.9	$1.1\times^{\dagger}$	$1.1\times^{\dagger}$	1,205.1	$1.1\times^{\dagger}$	$1.1\times^{\dagger}$
9	seq-rd-32th-32f	4	2.6	$1.5\times^{\dagger}$	$2.4\times^{*}$	7.2	$2.5\times^{*}$	$2.4\times^{*}$
10		32	18.3	$1.5\times^{\dagger}$	$2.6\times^{*}$	48.5	$2.7\times^{*}$	$2.7\times^{*}$
11		128	72.6	$1.6\times^{\dagger}$	$2.6\times^{*}$	192.1	$2.8\times^{*}$	$2.7\times^{*}$
12		1,024	561.4	$1.6\times^{\dagger}$	$2.6\times^{*}$	1,494.9	$2.8\times^{*}$	$2.7\times^{*}$
13	rnd-rd-1th-1f	4	36.8	$2.1\times^{*}$	$3.6\times^{\#}$	4.2	$2.5\times^{*}$	$4.6\times^{\#}$
14		32	49.0	$2.1\times^{*}$	$3.1\times^{\#}$	40.9	$2.2\times^{*}$	$2.8\times^{*}$
15		128	91.0	$2.5\times^{*}$	$2.8\times^{*}$	164.8	$2.1\times^{*}$	$2.3\times^{*}$
16		1,024	1,043.6	$2.4\times^{*}$	$2.8\times^{*}$	1,335	$2.2\times^{*}$	$2.8\times^{*}$
17	rnd-rd-32th-1f	4	44.7	$1.6\times^{\dagger}$	$4.5\times^{\#}$	30.9	$0.4\times^{\ddagger}$	$4\times^{\#}$
18		32	68.2	$1.5\times^{\dagger}$	$3.6\times^{\#}$	72.0	$1.3\times^{\dagger}$	$3.5\times^{\#}$
19		128	130.0	$1.7\times^{\dagger}$	$2.5\times^{*}$	210.5	$1.8\times^{\dagger}$	$2.5\times^{*}$
20		1,024	509.8	$1.6\times^{\dagger}$	$2.2\times^{*}$	1,482.3	$2.4\times^{*}$	$2.5\times^{*}$
21	seq-wr-1th-1f	4	17.7	$2.9\times^{*}$	$4.2\times^{\#}$	15.1	$3.7\times^{\#}$	$4.3\times^{\#}$
22		32	98.7	$4.2\times^{\#}$	$5.4\times^{\#}$	90.6	$4.8\times^{\#}$	$5.2\times^{\#}$
23		128	386.7	$4.3\times^{\#}$	$5.4\times^{\#}$	356.4	$4.8\times^{\#}$	$5.2\times^{\#}$
24		1024	3,132.0	$4.2\times^{\#}$	$5.4\times^{\#}$	2,877.8	$4.8\times^{\#}$	$5.3\times^{\#}$
25	seq-wr-32th-32f	4	4.6	$13.7\times^{!}$	$11.8\times^{!}$	4.5	$13.3\times^{!}$	$11.4\times^{!}$
26		32	32.3	$13.3\times^{!}$	$12.7\times^{!}$	30.5	$13.6\times^{!}$	$13\times^{!}$
27		128	132.1	$13.1\times^{!}$	$12.4\times^{!}$	124.5	$13.2\times^{!}$	$12.8\times^{!}$
28		1,024	1,036.7	$13.2\times^{!}$	$12.6\times^{!}$	1,007.0	$13\times^{!}$	$12.4\times^{!}$
29	rnd-wr-1th-1f	4	5.2	$3.2\times^{\#}$	$12.8\times^{!}$	5.8	$5.6\times^{\#}$	$11.5\times^{!}$
30		32	12.5	$10.1\times^{\#}$	$5.5\times^{\#}$	69.9	$6.2\times^{\#}$	$6.6\times^{\#}$
31		128	43.5	$11.7\times^{!}$	$10.6\times^{\#}$	229.1	$7.3\times^{\#}$	$7\times^{\#}$
32		1,024	686.5	$11.4\times^{!}$	$10.5\times^{\#}$	1,767.6	$7.4\times^{\#}$	$7\times^{\#}$
33	rnd-wr-32th-1f	4	5.4	$3.8\times^{\#}$	$18.6\times^{!}$	28.2	$1.4\times^{\dagger}$	$3\times^{*}$
34		32	11.5	$11\times^{!}$	$9\times^{\#}$	107.3	$4.3\times^{\#}$	$4.6\times^{\#}$
35		128	39.6	$13.2\times^{!}$	$12.6\times^{!}$	335.7	$5.3\times^{\#}$	$5.1\times^{\#}$
36		1,024	685.9	$11.7\times^{!}$	$11.4\times^{!}$	1,813.5	$7.6\times^{\#}$	$7.5\times^{\#}$
37	files-cr-1th	4	213.9	$1.2\times^{\dagger}$	$0.8\times^{\ddagger}$	241.2	$1.1\times^{\dagger}$	$0.8\times^{\ddagger}$
38	files-cr-32th	4	743.8	$0.5\times^{\ddagger}$	$0.7\times^{\ddagger}$	938.0	$0.4\times^{\ddagger}$	$0.6\times^{\ddagger}$
39	files-del-1th	4	27.0	$2.4\times^{*}$	$3.4\times^{\#}$	85.0	$1.8\times^{\dagger}$	$1.6\times^{\dagger}$
40	files-del-32th	4	50.7	$1.4\times^{\dagger}$	$1.8\times^{\dagger}$	223.1	$0.7\times^{\ddagger}$	$0.8\times^{\ddagger}$

(Continued)

Table 8. Continued

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			Ext4 (ns/loop)	SBase (factor)	SOpt (factor)	Ext4 (ns/loop)	SBase (factor)	SOpt (factor)
41	files-rd-1th	-	76.7	2.2×*	3.5×#	189.6	2.3×*	2.1×*
42	files-rd-32th	-	114.9	1.9×†	3.7×#	921.7	0.6×‡	3×#
43	file-server	-	1,240.0	2.3×*	3.3×#	1,223.0	1.9×†	3.8×#
44	mail-server	-	1,035.0	1.5×†	2.9×*	1,423.4	0.6×‡	2.1×*
45	web-server	-	2,459.3	1.5×†	3.3×#	2,407.2	0.8×‡	5×#

(2) The *Green* class (marked with †) indicates that the difference in CPU utilization is in the range of (0, 1×].
(3) The *Yellow* class (*) indicates that the difference in CPU utilization is in the range of (1×, 2×].
(4) The *Orange* class (#) indicates that the difference in CPU utilization is in the range of (2×, 10×].
(5) And finally, the *Red* class (!) indicates that difference in CPU utilization is > 10×.

Below, we list our main observations drawn from the CPU utilization results using Table 8.

■ Observation 1. The relative increase in the CPU utilization varied across workloads, devices, and FUSE configurations from about 0.5× for `files-cr-32th` [Table 8, row #38] to 12.5× `sq-wr-32th-32f` [Table 8, row #25].

■ Observation 2. As expected, for most workloads SBase increased the CPU utilization, except for a few cases such as `files-cr-32th` [Table 8, row #38], SSD in `files-del-32th` [Table 8, row #40], SSD in `mail-server` [Table 8, row #44], SSD in `web-server` [Table 8, row #45].

■ Observation 3. For most workloads, FUSE's optimizations increased the CPU utilization, except for a few cases such as `files-cr-1th` [Table 8, row #37], `files-cr-32th` [Table 8, row #38], and SSD in `files-del-32th` [Table 8, row #40].

■ Observation 4. There is no difference in the CPU utilization as I/O size varies in individual workloads.

■ Observation 5. Among all the workloads, `seq-wr-32th-32f`, `rnd-wr-1th-1f` and `rnd-wr-32th-1f` are the ones that have much higher CPU utilization for SBase and SOpt (i.e., almost all are in Red or Orange classes, peaking at 18.6× CPU utilization increase compared to Ext4, [Table 8, rows #25–36]).

■ Observation 6. At least in one Stackfs configuration, SOpt, all write workloads (sequential and random) [Table 8, rows #21–36] fall within the Orange and Red classes (except for [Table 8, row #33]).

■ Observation 7. SBase performs worse for metadata-intensive and macro workloads [Table 8, rows #37–45] than for data-intensive workloads [Table 8, rows #1–36], especially in SSDs.

■ Observation 8. Among all the workloads, only the `seq-wr-32th-32f` belongs to the Red class for both HDD and SSD for both SBase and SOpt [Table 8, rows #25–28].

Disk Bandwidth Utilization. In our experiments, we first measured disk bandwidth in MB/sec and then normalized it by the maximum possible disk bandwidth achieved with a sequential I/O on a raw device without any file system. Table 9 shows the results for the normalized disk bandwidth

Table 9. List of Workloads and Corresponding Normalized Disk-Bandwidth Results

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			Ext4 (%)	SBase (%Diff)	SOpt (%Diff)	Ext4 (%)	SBase (%Diff)	SOpt (%Diff)
1	seq-rd-1th-1f	4	96	$-1.0^{\dagger}$	$-1.0^{\dagger}$	89	$0.0^{\ddagger}$	$0.0^{\ddagger}$
2		32	96	$-1.0^{\dagger}$	$-1.0^{\dagger}$	89	$0.0^{\ddagger}$	$0.0^{\ddagger}$
3		128	96	$-1.0^{\dagger}$	$-1.0^{\dagger}$	89	$0.0^{\ddagger}$	$+1.1^{\ddagger}$
4		1,024	96	$-1.0^{\dagger}$	$-1.0^{\dagger}$	89	$0.0^{\ddagger}$	$0.0^{\ddagger}$
5	seq-rd-32th-1f	4	96	$0.0^{\ddagger}$	$-1.0^{\ddagger}$	89	$0.0^{\ddagger}$	$+2.2^{\ddagger}$
6		32	96	$-1.0^{\dagger}$	$-2.1^{\dagger}$	89	$0.0^{\ddagger}$	$+1.1^{\ddagger}$
7		128	96	$-1.0^{\dagger}$	$-2.1^{\dagger}$	89	$0.0^{\ddagger}$	$+1.1^{\ddagger}$
8		1,024	96	$-1.0^{\dagger}$	$-1.0^{\dagger}$	89	$0.0^{\ddagger}$	$0.0^{\ddagger}$
9	seq-rd-32th-32f	4	28	$-35.7^{\#}$	$-28.6^{\#}$	95	$0.0^{\ddagger}$	$0.0^{\ddagger}$
10		32	30	$-43.3^{\#}$	$-30.0^{\#}$	98	$0.0^{\ddagger}$	$-1.0^{\dagger}$
11		128	30	$-43.3^{\#}$	$-30.0^{\#}$	98	$0.0^{\ddagger}$	$-1.0^{\dagger}$
12		1,024	30	$-43.3^{\#}$	$-30.0^{\#}$	98	$0.0^{\ddagger}$	$-1.0^{\dagger}$
13	rnd-rd-1th-1f	4	1	$0.0^{\ddagger}$	$0.0^{\ddagger}$	13	$-30.8^{\#}$	$-38.5^{\#}$
14		32	4	$0.0^{\ddagger}$	$0.0^{\ddagger}$	45	-17.8^{*}	-24.4^{*}
15		128	14	$0.0^{\ddagger}$	$0.0^{\ddagger}$	75	-13.3^{*}	-12.0^{*}
16		1,024	59	$+3.4^{\ddagger}$	$0.0^{\ddagger}$	89	$-3.4^{\dagger}$	$0.0^{\ddagger}$
17	rnd-rd-32th-1f	4	1	$0.0^{\ddagger}$	$0.0^{\ddagger}$	69	$-82.6^{!}$	-24.6^{*}
18		32	10	$-60.0^{!}$	-20.0^{*}	94	$-55.3^{!}$	$-1.1^{\dagger}$
19		128	21	$-33.3^{\#}$	-9.5^{*}	98	$-27.6^{\#}$	$0.0^{\ddagger}$
20		1,024	27	$-33.3^{\#}$	-11.1^{*}	98	-9.2^{*}	$0.0^{\ddagger}$
21	seq-wr-1th-1f	4	92	$-26.1^{\#}$	$-1.1^{\dagger}$	95	-7.4^{*}	$0.0^{\ddagger}$
22		32	92	-17.4^{*}	$-1.1^{\dagger}$	95	$-2.1^{\dagger}$	$0.0^{\ddagger}$
23		128	92	-16.3^{*}	$-1.1^{\dagger}$	95	$-1.0^{\dagger}$	$0.0^{\ddagger}$
24		1,024	92	-17.4^{*}	$-1.1^{\dagger}$	95	$-1.0^{\dagger}$	$0.0^{\ddagger}$
25	seq-wr-32th-32f	4	86	$-2.3^{\dagger}$	$-1.2^{\dagger}$	95	$0.0^{\ddagger}$	$0.0^{\ddagger}$
26		32	86	$-2.3^{\dagger}$	$-1.2^{\dagger}$	95	$0.0^{\ddagger}$	$0.0^{\ddagger}$
27		128	85	$-1.2^{\dagger}$	$0.0^{\ddagger}$	95	$0.0^{\ddagger}$	$0.0^{\ddagger}$
28		1,024	85	$-1.2^{\dagger}$	$0.0^{\ddagger}$	95	$0.0^{\ddagger}$	$0.0^{\ddagger}$
29	rnd-wr-1th-1f	4	2	$0.0^{\ddagger}$	$0.0^{\ddagger}$	46	$0.0^{\ddagger}$	$-26.1^{\#}$
30		32	14	$0.0^{\ddagger}$	$0.0^{\ddagger}$	93	$0.0^{\ddagger}$	-9.7^{*}
31		128	28	$0.0^{\ddagger}$	$0.0^{\ddagger}$	95	$0.0^{\ddagger}$	$0.0^{\ddagger}$
32		1,024	50	$0.0^{\ddagger}$	$0.0^{\ddagger}$	95	$0.0^{\ddagger}$	$-1.0^{\dagger}$
33	rnd-wr-32th-1f	4	2	$0.0^{\ddagger}$	$0.0^{\ddagger}$	46	$0.0^{\ddagger}$	$-26.1^{\#}$
34		32	14	$0.0^{\ddagger}$	$0.0^{\ddagger}$	93	$0.0^{\ddagger}$	-12.9^{*}
35		128	28	$0.0^{\ddagger}$	$0.0^{\ddagger}$	95	$0.0^{\ddagger}$	$0.0^{\ddagger}$
36		1,024	50	$0.0^{\ddagger}$	$-2.0^{\dagger}$	95	$0.0^{\ddagger}$	$-1.0^{\dagger}$
37	files-cr-1th	4	31	$-58.1^{!}$	$-80.6^{!}$	42	$-61.9^{!}$	$-83.3^{!}$
38	files-cr-32th	4	37	$-48.6^{\#}$	$-59.5^{!}$	54	$-57.4^{!}$	$-61.1^{!}$
39	files-del-1th	4	2	$0.0^{\ddagger}$	$-50.0^{!}$	21	-23.8^{*}	$-57.1^{!}$
40	files-del-32th	4	3	$-33.3^{\#}$	$0.0^{\ddagger}$	66	$-74.2^{!}$	$-33.3^{\#}$
41	files-rd-1th	-	5	$0.0^{\ddagger}$	$0.0^{\ddagger}$	42	$-33.3^{\#}$	$-61.9^{!}$

(Continued)

Table 9. Continued

#	Workload	I/O Size (KB)	HDD Results			SSD Results		
			Ext4 (%)	SBase (%Diff)	SOpt (%Diff)	Ext4 (%)	SBase (%Diff)	SOpt (%Diff)
42	files-rd-32th	-	5	$0.0^{\ddagger}$	$0.0^{\ddagger}$	48	$-39.6^{\#}$	$-54.2^{!}$
43	file-server	-	25	-24.0^{*}	$0.0^{\ddagger}$	95	$-41.0^{\#}$	$-2.1^{\dagger}$
44	mail-server	-	11	$-45.4^{\#}$	-9.1^{*}	76	$-60.5^{!}$	-17.1^{*}
45	web-server	-	7	$-42.9^{\#}$	$0.0^{\ddagger}$	81	$-74.1^{!}$	-9.9^{*}

in % for Ext4 and relative performance for two Stackfs configurations for both the HDD and SSD. For instance, in [Table 9, row #10], the normalized disk bandwidth for Ext4 is 30%. Relative to it, the disk bandwidth of SBase is −43%, which means that the absolute normalized disk bandwidth of SBase is 17.1%. In [Table 9, row #18], the normalized disk bandwidth of Ext4 is 10%. Relative to it, the normalized disk bandwidth of SBase is −60% and SOpt is −20%. So, the absolute normalized disk bandwidth of SBase is 4% and SOpt is 8%. We categorized the results (from Table 9) by Stackfs's normalized disk bandwidth (relative to Ext4) into five classes:

(1) The *Blue* class (marked with ‡) indicates that there is an improvement in the normalized disk bandwidth of Stackfs over Ext4.
(2) The *Green* class (marked with †) indicates that the degradation in the normalized disk bandwidth is <5%.
(3) The *Yellow* class (*) indicates that the degradation in the normalized disk bandwidth is in the [5–25)% range.
(4) The *Orange* class ($^{\#}$) indicates that the degradation in the normalized disk bandwidth is in the [25–50)% range.
(5) The *Red* class ($^{!}$) indicates that the degradation in the normalized disk bandwidth is >50%.

Below, we list our main observations drawn from the normalized disk bandwidth results using Table 9.

■ Observation 1. The relative increase in the normalized disk bandwidth varied across workloads, devices, and FUSE configurations from −82.6% in SBase for SSD in `rnd-rd-32th-1f` [Table 9, row #17] to 3.4% in SBase for HDD in `rnd-rd-1th-1f` [Table 9, row #16].

■ Observation 2. For almost half of the workloads, in each of SBase and SOpt for both HDD and SSD, we saw the same or a marginal increase in normalized disk bandwidth.

■ Observation 3. The normalized disk bandwidth for the data-intensive workloads, in SBase and SOpt, grows as I/O sizes increase in each individual workload, for both HDD and SSD [Table 9, rows #1–45].

■ Observation 4. Among all the workloads, `files-cr-1th` [Table 9, row #37] and `files-cr-32th` [Table 9, row #38] are the ones which have much lower relative normalized disk bandwidth in SBase and SOpt (i.e., almost all are in the Red class).

■ Observation 5. Stackfs's disk utilization depends significantly on the underlying device. For example, in seq-rd-32th-32f workloads [Table 9, rows #9–12], Stackfs shows almost no performance degradation for SSD and a 30–43% degradation for HDD. The situation is reversed, for example, for the `rnd-rd-1th-1f` workload [Table 9, rows #13–16], the `files-rd-1th` workload [Table 9, row #41], or the `files-rd-32th` workload [Table 9, row #42].

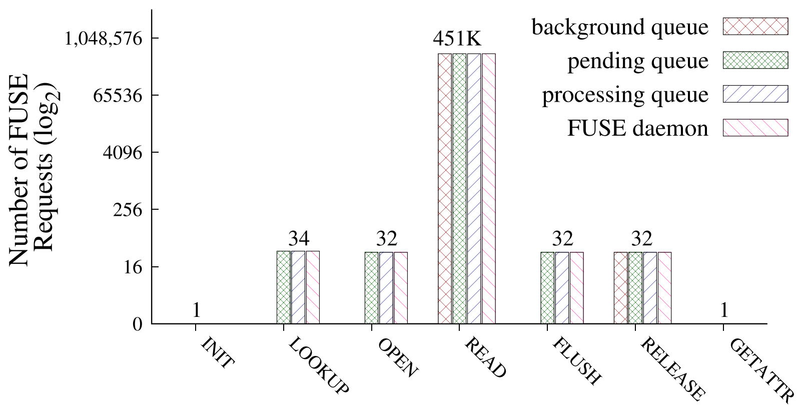

Fig. 6. Different types and number of requests generated by SBase on SSD during the `seq-rd-32th-32f` workload, from left to right, in their order of generation.

■ Observation 6. In sequential read workloads [Table 9, rows #1–12], SBase and SOpt for SSD showed almost no degradation or improvement in the relative normalized disk bandwidth. The same applies to SBase for SSD in the write workloads [Table 9, rows #21–36].

■ Observation 7. In general, SBase and SOpt underutilized disk bandwidth more for metadata-intensive and macro workloads [Table 9, rows #37–45] than for data-intensive workloads [Table 9, rows #1–36], especially for SSD.

5.2 Detailed Analysis

We analyzed FUSE performance results and present main findings here, following the order in Table 6. The next subsections detail read workloads, write workloads, meta-data workloads, macro workloads, and the impact of performance vs. resource utilization; finally, the impact of individual FUSE optimizations is described in Section 5.2.6.

5.2.1 Read Workloads. Figure 6 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 [Table 6, 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; the 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 6 also shows the breakdown of requests across queues. By default, READ, RELEASE, and INIT are asynchronous requests. Therefore, they are added to the background queue first, whereas all other requests are synchronous and are added to the pending queue directly. In read workloads, only READ requests are generated in large numbers. Therefore, we consider only READ requests when we discuss each workload in detail.

For all the read-intensive workloads [Table 8, rows #1–20], the CPU utilization per loop, of SBase and SOpt falls under the *Green*, *Yellow*, and *Orange* classes.

Table 10. List of Workloads and Corresponding Performance Results

#	Workload	I/O Size (KB)	HDD Results			
			Ext4 (ops/s)	SBase (%Diff)	SOpt (%Diff)	SOpt-100 (%Diff)
1	seq-rd-32th-32f	4	11,141	$-36.9^{\#}$	$-26.9^{\#}$	$+0.2^{\ddagger}$
2		32	1,491	$-41.5^{\#}$	$-30.3^{\#}$	$-1.0^{\dagger}$
3		128	371	$-41.3^{\#}$	$-29.8^{\#}$	$-2.7^{\dagger}$
4		1,024	46	$-41.0^{\#}$	$-28.3^{\#}$	$-2.0^{\dagger}$

SOpt-100 refers to SOpt with background_queue limit increased to 100.

Sequential Read Using 1 Thread on 1 File. Surprisingly, the total number of READ requests SBase generated during the whole experiment for different I/O sizes for HDD and SSD 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 normalizes FUSE request sizes no matter what the user application I/O size is. Thanks to readahead, sequential read performance of SBase and SOpt was as good as Ext4 for both HDD and SSD.

Sequential Read Using 32 Threads on 32 Files. Due to readahead, the total number of READ requests generated here was also approximately the same for different I/O sizes. At any given time, 32 threads were requesting data and continuously add requests to queues. SBase and SOpt show significantly larger performance degradation on HDD compared to SSD. For SBase, the FUSE 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 FUSE daemon is single threaded, requests move faster in the queues. Hence, performance of SBase is as close to that of Ext4. With SOpt, the FUSE 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 we investigated further, we found that in the case of HDD and SOpt, the FUSE daemon was bound by the *max_background* value (default is 12): at most only 12 FUSE daemons (threads) were invoked. So we increased that limit to 100 and reran the experiments. Table 10 shows the improved results, which demonstrate that SOpt 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 the number of requests. This modest increase is because, in the beginning of the experiment, every thread tries to read the data separately; but after a certain point in time, only a single thread's requests are propagated to the FUSE 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 FUSE 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, the more READ requests are generated and the higher FUSE's overhead tended to be. This is seen for SOpt 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, SOpt performs

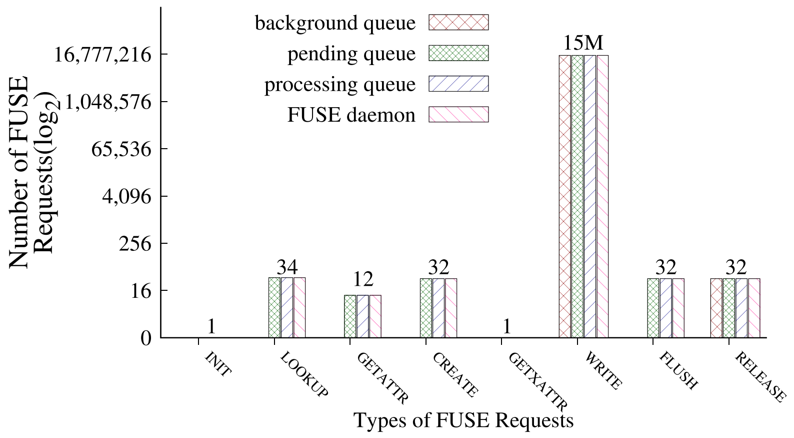

Fig. 7. Different types of requests that were generated by SBase on SSD for the `seq-wr-32th-32f` workload, from left to right in their order of generation.

better than SBase 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 FUSE daemon and causes high performance degradation: up to −83% for SBase and −28% for SOpt. The overhead caused by SBase 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 FUSE daemon processes one request at a time. SOpt reduced performance degradation compared to SBase, but not as much for 4KB I/Os because splice is not used for requests that are smaller or equal to 4KB.

5.2.2 Write Workloads. We now discuss the behavior of SBase and SOpt in all write workloads listed in Table 6 [Table 6, rows #21–36]. Figure 7 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 the 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, SBase and SOpt did not support extended attributes and the kernel cached this knowledge after FUSE returned `ENOSUPPORT` for the first GETXATTR. For almost all write-intensive workloads [Table 8, rows #21–36], the CPU utilization per loop, of SBase and SOpt falls into the *Red* and *Orange* classes.

Sequential Write Using 1 Thread on 1 File. The total number of WRITE requests that SBase generated during this experiment was 15.7 million for all I/O sizes. This is because in SBase each user write call is split into several 4KB-size FUSE requests which are sent to the FUSE daemon. As a result, SBase degraded performance ranged from −26% to −2%. Compared to SBase, SOpt 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 SOpt compared to SBase. 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 SBase performed significantly better (only up to −2.7% and −0.1% 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 FUSE daemon constantly busy.

Random Write Using 1 Thread on 1 File. The performance degradation caused by SBase and SOpt 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 1,074 Filebench ops/sec, depending on the I/O size (compare to over 16,000 ops/sec for SSD). The performance bottleneck, therefore, was in the HDD I/O time and FUSE overhead was not visible.

Interestingly, on SSD, SOpt performance degradation was high (−27% for 4KB I/O) and more than the SBase 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 SBase. 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 an I/O size of 32KB, SOpt can pack the entire 32KB into a single WRITE request. Compared to SBase, this reduces the number of WRITE requests by 8× and results in 15% better performance.

Random Write Using 32 Threads on 1 File. This workload performs similarly to `rnd-wr-1th-1f` and the same analysis applies.

5.2.3 Metadata Workloads. We now discuss the behavior of SBase and SOpt in all metadata micro-workloads as listed in Table 6 [Table 6, rows #37–42].

File Creates. Figure 8 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 SBase and SOpt 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 overheads caused by the FUSE requests' user-kernel communication explain the performance degradation.

File Reads. Figure 9 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 9 shows that half of the READ requests went to the background queue and the other half directly to the pending queue. The reason for this is that when reading a whole file, and the application requests reads beyond the EOF, FUSE generates a synchronous READ request which goes to the pending queue (instead of

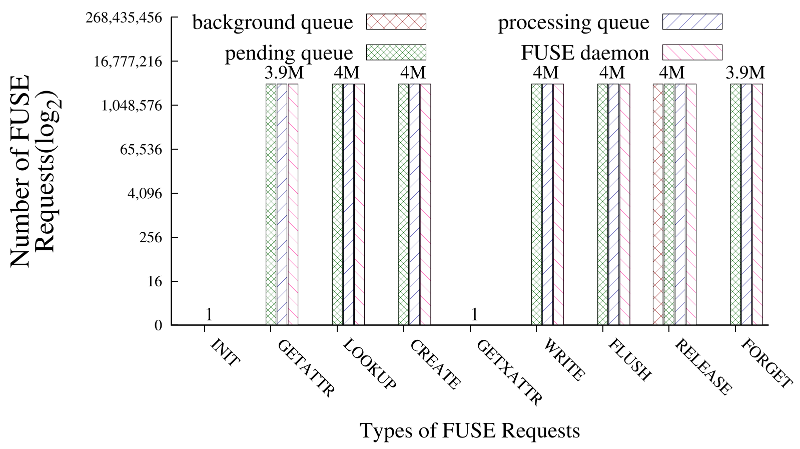

Fig. 8. Different types of requests that were generated by SBase on SSD for the `files-cr-1th` workload, from left to right in their order of generation.

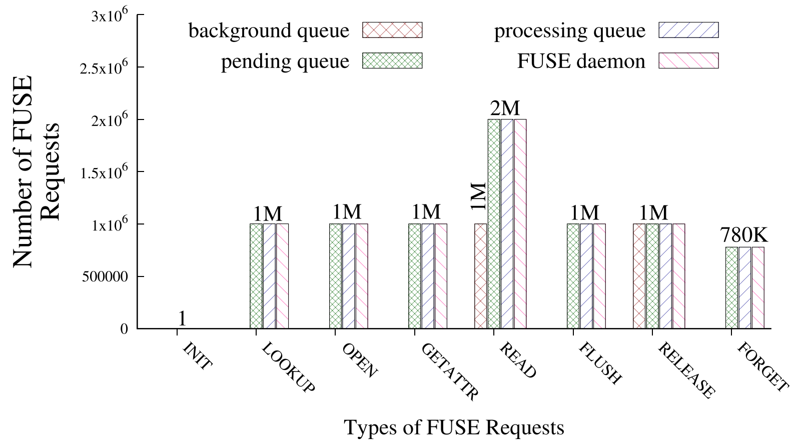

Fig. 9. Different types of requests that were generated by SBase on SSD for the `files-rd-1th` workload, from left to right in their order of generation.

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 SBase on HDD is negligible and close to Ext4; on SSD, however, the relative degradation is high (–25%) because the SSD is 12.5× faster than HDD (see Ext4 absolute throughput in Table 6). Interestingly, SOpt's performance degradation is more than that of SBase (by 10% and 35% for HDD and SSD, respectively). The reason is that in SOpt, *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, SOpt performed 40–45% better than SBase because multiple threads are needed to effectively process parallel READ requests.

File Deletes. Figure 10 shows different types of operations that got generated during the `files-del-1th` workloads. Every UNLINK request is followed by a FORGET request. Therefore, for every

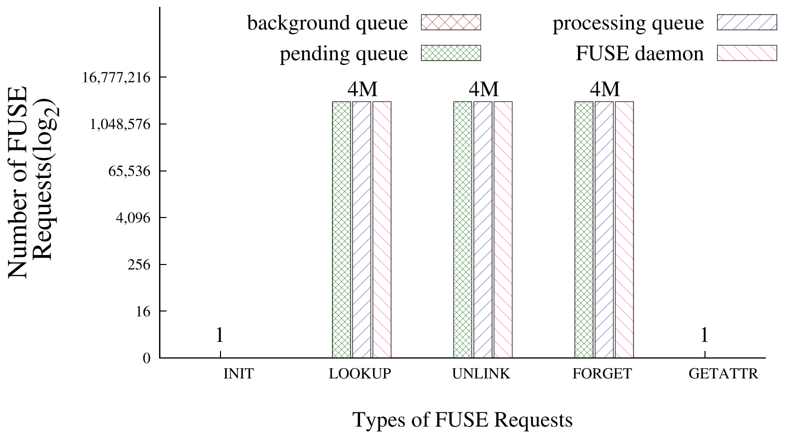

Fig. 10. Different types of requests that were generated by SBase on SSD for the `files-del-1th` workload, from left to right in their order of generation.

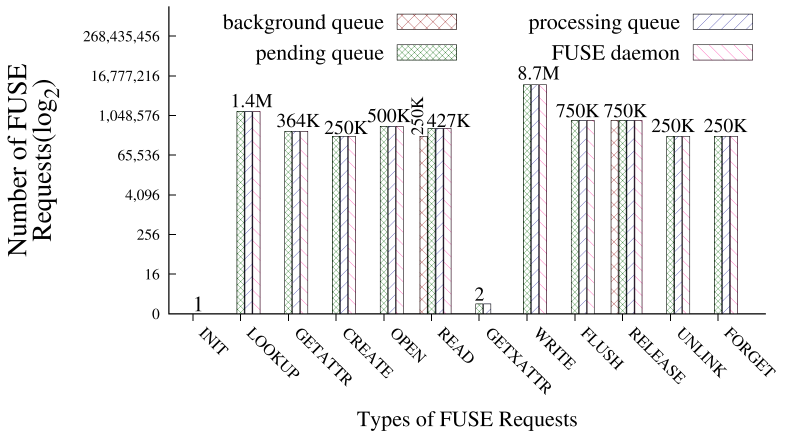

Fig. 11. Different types of requests that were generated by SBase on SSD for the `file-server` workload.

incoming delete request that the application (Filebench) submits, SBase and SOpt 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 SOpt's performance degradation is more than SBase's because neither splice nor the writeback cache helped `files-del-`*N*`th` workloads and only added additional overhead for managing extra threads.

5.2.4 Macro Server Workloads. We now discuss the behavior of Stackfs for macro-workloads [Table 6, rows #43–45].

File Server. Figure 11 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.

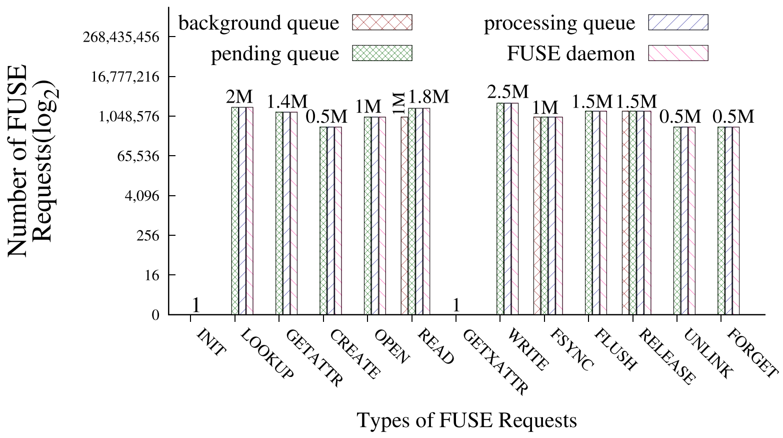

Fig. 12. Different types of requests that were generated by SBase on SSD for the `mail-server` workload.

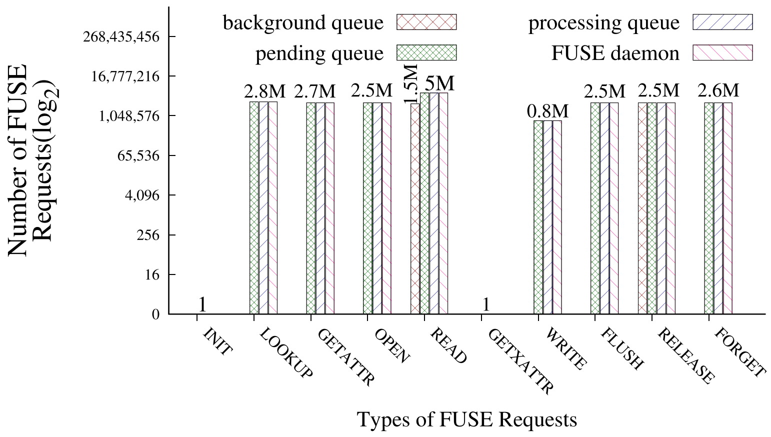

Fig. 13. Different types of requests that were generated by SBase on SSD for the `web-server` workload.

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

Mail Server. Figure 12 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 SOpt 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 meta-data operations, which increases FUSE overhead. Second, I/O sizes are smaller in mail-server which improves the underlying Ext4 SSD performance and therefore shifts the bottleneck to FUSE.

Web Server. Figure 13 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

Table 11. Consolidated Table with the Performance and the Resource Utilization Parameters

Parameter	**HDD Results**			**SSD Results**		
	Ext4	**SBase**	**SOpt**	**Ext4**	**SBase**	**SOpt**
seq-rd-32th-32f-4K						
Throughput (ops/sec)	11,141	$-36.9^{\#}$	$-26.9^{\#}$	32,855	$-0.1^{\dagger}$	$-0.2^{\dagger}$
Latency (ms/op)	2.91	$+56.2^{!}$	$+33.7^{\#}$	0.97	$-0.1^{\ddagger}$	$+0.1^{\dagger}$
Normalized Bandwidth (%)	28	$-35.7^{\#}$	$-28.6^{\#}$	95	$0.0^{\ddagger}$	$0.0^{\ddagger}$
CPU utilization (nsec/loop)	2.6	$1.5\times^{\dagger}$	$2.4\times^{*}$	7.2	$2.5\times^{*}$	$2.4\times^{*}$
seq-rd-32th-32f-32K						
Throughput (ops/sec)	1,491	$-41.5^{\#}$	$-30.3^{\#}$	4,202	$-0.1^{\dagger}$	$-1.8^{\dagger}$
Latency (ms/op)	21.44	$+72.8^{!}$	$+17.6^{*}$	7.58	$-0.1^{\ddagger}$	$+0.1^{\dagger}$
Normalized Bandwidth (%)	30	$-43.3^{\#}$	$-30.0^{\#}$	98	$0.0^{\ddagger}$	$-1.0^{\dagger}$
CPU utilization (nsec/loop)	18.3	$1.5\times^{\dagger}$	$2.6\times^{*}$	48.5	$2.7\times^{*}$	$2.7\times^{*}$

Web-pages. The performance degradation caused by SBase 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 SOpt significantly on both HDD and SSD, mainly thanks to using multiple threads. In fact, SOpt performance on HDD is even 6% *higher* than of native Ext4. We believe this minor improvement is caused by the Linux VFS treating Stackfs and Ext4 as two independent file systems and allowing them together to cache more data compared to when Ext4 is used alone, without Stackfs. This does not help SSD setup as much due to the high speed of SSD.

5.2.5 Performance vs. Resource Utilization. In previous sections, we looked at various metrics—throughput, latency, CPU, and disk utilization—*independently* across all evaluated workloads. In this section, we reorganize the numbers already presented by consolidating *all* metrics for a specific workload in one place. We believe that such representation is useful for people interested in knowing the *overall* impact of FUSE on a specific workload. For brevity, we demonstrate this approach only for two workloads: `seq-rd-32th-32f-4K` and `seq-rd-32th-32f-32K` but similar analysis can be performed for other workloads based on the results presented in earlier sections.

Table 11 consolidates the following information, for the `seq-rd-32th-32f-4K` and `seq-rd-32th-32f-32K` workloads, for two Stackfs configurations on HDD and SSD:

—throughput (in ops/sec for Ext4 and in percents of relative degradation for Stackfs);
—latency (in ms/op for Ext4 and in percents of relative degradation or improvement for Stackfs);
—CPU utilization (nsec/loop for Ext4 and in factors of relative degradation for Stackfs).
—Disk bandwidth (in percentage, normalized to the maximum possible disk bandwidth, for Ext4 and in percents of relative degradation for Stackfs).

The latency and normalized disk bandwidth follow a similar behavior as the throughput. Considering `seq-rd-32th-32f-4K`, at any given time, 32 threads were requesting data and continuously adding requests to queues. For SBase on HDD, the FUSE daemon is single-threaded and the device is slower, so requests do not move through quickly; hence the throughput is lower by ~37% than that of Ext4. This is also reflected in the normalized disk bandwidth which decreases by approximately the same amount, ~36%. For SOpt on HDD, the FUSE daemon is multi-threaded, but can have at most 12 FUSE daemons (threads), which is an implementation limit. The multi-threading is not 1-to-1 (meaning that the workload's threads are independent of FUSE's threads). As a result, the latency increases by ~34%, due to extra time spent in resolving contention among the user threads for the FUSE daemons. It can be observed that the throughput and normalized disk bandwidth also decrease, by approximately the same amount, ~27% and ~29%, respectively.

On the faster SSD, even though the FUSE daemon is single threaded, requests move faster; hence, the similar variations observed in throughput, latency, and normalized disk bandwidth, are much smaller here (0–0.2%). Considering `seq-rd-32th-32f-32K`, the throughput, latency, and CPU utilization of Ext4 show a variation relative to that of `seq-rd-32th-32f-4K`, consistent with the size of I/O: that is, throughput decreases, latency decreases, and CPU utilization increases 8×. For SBase and SOpt on HDD, the throughput and normalized disk bandwidth show the same amount of decrease: by ~42% for SBase and ~30% for SOpt. The latency shows an increase, though by a different amount: 72.8% for SBase and 17.6% for SOpt. For SBase and SOpt on SSD, as in the case of `seq-rd-32th-32f-4K`, the variations in throughput, latency, and normalized disk bandwidth are similar and very small (0–2%).

Notice that latency numbers are consistent with the throughput. For example, for Ext4 and `seq-rd-32th-32f-4K` workload, throughput and latency are 11,142ops/sec and 2.91ms/op, respectively. By computing latency from the throughput using ($\frac{1}{11,142} \times 32$), where 32 is the number of threads in the workload, we obtain 2.87ms/op—close to the measured 2.91ms/op latency.

The CPU utilization does not follow a percentage of degradation or improvement similar to that of throughput, latency, or the normalized disk bandwidth, in the above two workloads. It is not surprising, as CPU utilization is orthogonal to other performance metrics. But it does show a percentage degradation (increase) for SBase and SOpt on HDD and SSD. In the case of `seq-rd-32th-32f-4K` and `seq-rd-32th-32f-32K` on HDD, the CPU utilization of SBase is more than that of Ext4 due to the communication overhead of the FUSE layer. The CPU utilization of SOpt is more than that of SBase, due to the impact of multiple threads utilizing more CPU time. In either `seq-rd-32th-32f-4K` or `seq-rd-32th-32f-32K`, the CPU utilization of Ext4 and SBase on SSD is more than the respective ones on HDD, due to a lower latency of the device. The CPU utilization of SOpt on SSD is similar to that on HDD, because the effect of the optimizations outweighs that of the latency of the device.

5.2.6 Impact of Individual Optimizations. In the previous sections, we evaluated two coarse-grained FUSE configurations—SBase and SOpt. These configurations differ in several parameter values, such as the presence or absence of a write cache 2.9, multi-threading 2.8, or splice 2.6. These parameters are explained in detail in Section 2. This section evaluates the impact of several individual FUSE optimizations on the throughput and CPU utilization. We present this analysis for a subset of workloads to illustrate the most interesting findings.

For the `seq-wr-128kb-1th-1f` workload, Figure 14 shows the throughput, for Ext4, SOpt, and SBase with a combination of FUSE optimizations; Figure 15 shows the CPU utilization, for Ext4, SOpt, and SBase with a combination of FUSE optimizations. In the X-axis labels corresponding to the combination of optimizations, WBC stands for writeback_cache, MT stands for multi-threading, and BMW stands for big_writes and max_write, with the max_write value set to 128KB. We observed the following in the throughput and the CPU utilization (going from left to right in Figures 14 and 15).

- SBase has a lower throughput and a higher CPU utilization than Ext4.
  - Communication overhead from the FUSE kernel, via the FUSE Daemon, and on to Ext4, and having smaller 4KB write chunks, all contributed to this.
- No improvement is seen in SBase if splice is enabled.

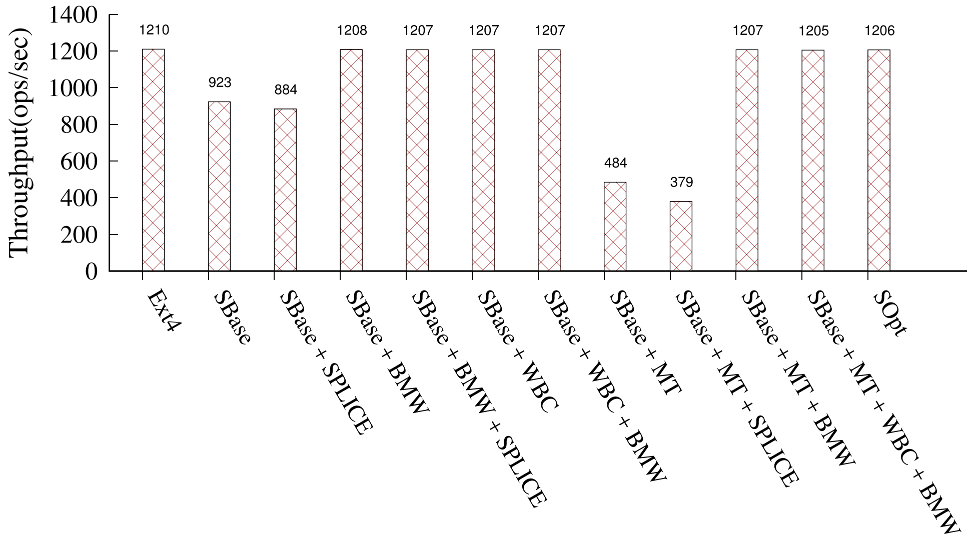

Fig. 14. Impact of incremental optimizations on throughput on HDD SEQ-WR-128KB-1TH-1F.

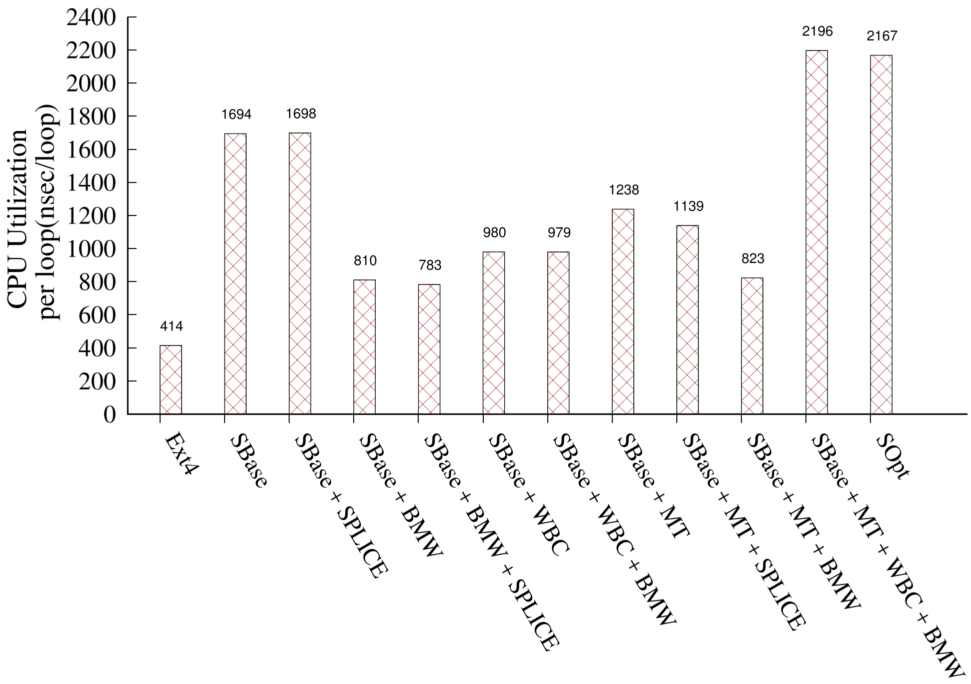

Fig. 15. Impact of incremental optimizations on CPU utilization per loop on HDD SEQ-WR-128KB-1TH-1F.

  — splice works only when FUSE requests are ≥8KB; and in the SBase case, 128KB user reads are split into multiple 4KB FUSE requests.

— Enabling big_writes avoids 4KB-chunking and shows a noticeable 35% improvement in throughput compared to SBase. It also shows a 52% lower CPU utilization over SBase, though still twice more than Ext4 due to communication overhead.

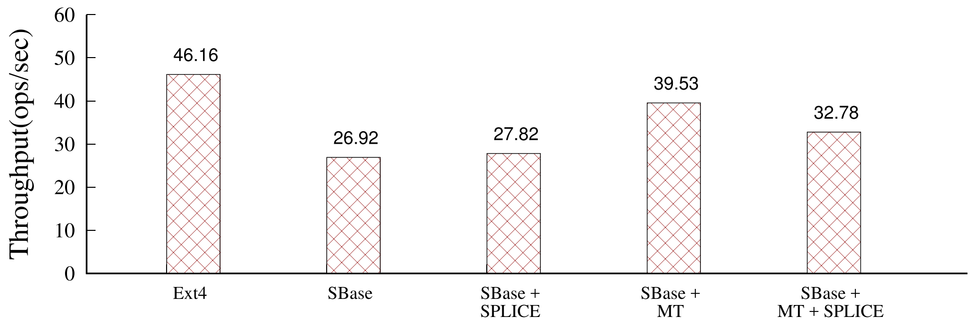

Fig. 16. Impact of incremental optimizations on throughput on HDD SEQ-RD-1MB-32TH-32F.

  - —But no difference is present between big_writes with splice and without splice, which implies that big_writes are highly dominant over splice in their impact. Furthermore, splice does not have an impact on the CPU utilization.
- —FUSE multi-threading leads to significantly lower throughput (2×) and a slightly lower CPU utilization (−26%) than SBase.
  - —Multi-threaded access to the same single file leads to CPU and I/O scheduling overheads, and increases file system operation latencies. This leads to a lower throughput.
  - —Splice further decreases the throughput and the CPU utilization per loop.
  - —However, when big_writes are enabled, their impact dominates the performance: multi-threaded configuration reaches the same throughput as a single-threaded FUSE daemon.
- —Writeback_cache has a similar impact as big_writes. Writeback cache allows FUSE to buffer data in the page cache to mask performance impact on the primary application thread. In essence, the data is moved to the FUSE daemon asynchronously, unlike the configurations without a writeback cache.
  - —This asynchronous activity in writeback_cache leads to a higher CPU utilization than compared to configurations without writeback_cache (but with big_writes).
- —In the case of a multi-threaded FUSE daemon, the CPU utilization of a configuration with big_writes and writeback_cache is about 2.5× that of a configuration with only big_writes. However, in the case of a single-threaded FUSE daemon, the CPU utilization of a configuration with big_writes and writeback_cache is similar to that of a configuration with only big_writes. This is expected because even though the write_back optimization allows the FUSE kernel to buffer more data for processing by the FUSE daemon, the daemon has to be multi-threaded to process available requests in parallel.

FUSE is often criticized for the additional memory copying it performs compared to in-kernel file systems. We were therefore interested in understanding if and by how much the splice optimization mitigates this issue. Figure 16 shows the throughput for Ext4 and SBase with a combination of optimizations for a read workload. We picked the `seq-rd-32th-32f` workload with an I/O size of 1MB, since this was one of the few read workloads where there was an appreciable difference between the performances of SBase and SOpt. We selected a large 1MB I/O size to see the benefits of splice when copying large memory regions. We tried the following FUSE optimization combinations: one without any optimizations; one with only multi-threading enabled; one with only splice enabled; and one with both multi-threading and splice enabled. Figure 17 shows the CPU utilization for the same combination of optimizations, for the same workload. In both figures, MT represents multi-threading.

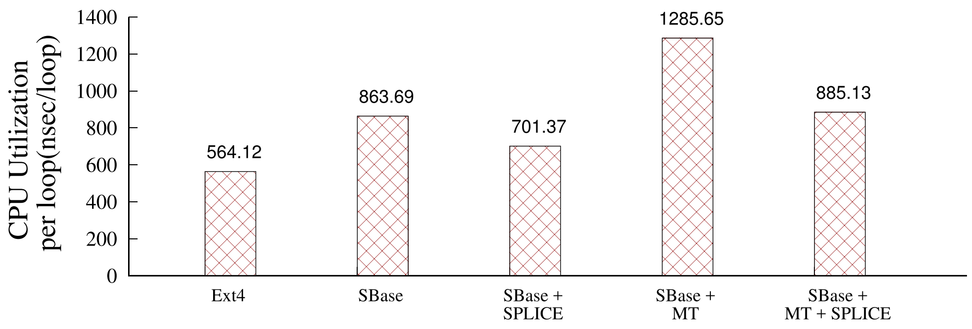

Fig. 17. Impact of incremental optimizations on CPU utilization per loop on HDD SEQ-RD-1MB-32TH-32F.

- All variations of the FUSE combinations have a lower throughput and a higher CPU utilization compared to Ext4.
  - As previously pointed out, this is due to the communication overhead of the FUSE layer. The CPU utilization in the FUSE configurations increases significantly due to this.
- Using a multi-threaded FUSE daemon (without splice enabled) results in an increase in both throughput and CPU utilization.
  - As the workload being used itself is multi-threaded, using a multi-threaded daemon also helps increase throughput. Conversely, the throughput goes down for the SBase experiment, since a single-threaded FUSE daemon has to handle 32 I/O threads.
  - More threads means more work, which leads to the observed higher CPU utilization.
- Enabling splice brings the CPU utilization down noticeably.
  - Since this is a 1MB workload, the effects of splice on CPU utilization is discernible. When used only with SBase or when used with SBase with multi-threading enabled, using splice results in a lower CPU utilization.
  - This effect is more apparent when the FUSE daemon is multi-threaded as the benefit of splice is distributed to more threads.
- However, splice has a negligible impact on throughput when enabled on top of SBase, and has a significant negative impact when used along with a multi-threaded FUSE daemon.

5.3 Evaluation Summary

Finally, we summarize all of our observations into just a few, in the hope of offering a succinct set of guidelines to those using or considering FUSE. Note that these guidelines follow directly from *this* study yet other users with different workloads may see different behavior. We present these observations from most to least obvious.

- For many workloads, clearly an optimized FUSE can perform better than a non-optimized FUSE.
- As expected, FUSE's performance depends significantly on the underlying device. If the underlying file system is naturally slow (e.g., network-bound storage such as SSHFS or S3FS), FUSE's performance or overheads may not matter much.
- A related observation is FUSE's overheads get worse with faster storage.
- FUSE-based file systems would generally require more CPU cycles than in-kernel file systems. Cycles are spent on FUSE's requests management, memory copying, and context switches. As a result, available disk bandwidth is expected to be less utilized by user-space file systems than by their kernel counterparts.
- For sequential, data-intensive workloads FUSE performs comparably well with Ext4.

—FUSE does not perform well for metadata-heavy workloads and workloads with small reads.
—Among data-intensive workloads, FUSE performs especially poorly for small random reads, irrespective of the underlying storage medium.
—However, for random writes, FUSE has none to minimal performance degradation.
—We enabled the following optimizations: Splice, writeback_cache, multi-threading, and BMW (big_writes and max_write), with the max_write value set to 128KB. We observed maximum throughput and minimum CPU utilization with BMW and Splice enabled for sequential write workloads.
—With only multi-threading enabled, maximum throughput is observed for sequential reads.

6 RELATED WORK

Many researchers used FUSE to implement file systems [8, 17, 28, 58] 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 [44]. Compared to our 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 optimization features. The authors also presented only limited information about FUSE design at that time.

Second, in a position paper, Tarasov et al. characterized FUSE performance for a variety of workloads but did not analyze the results [53]. Furthermore, they evaluated only default FUSE configuration and discussed only FUSE's high-level architecture. In this article, 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 [50]. To improve FUSE performance, Narayan et al. proposed to marry in-kernel stackable file systems [61] with FUSE [39]. Shun et al. modified FUSE's kernel module to allow applications to access storage devices directly [30]. These improvements were in research prototypes and were never included in the mainline.

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 article, we first presented the detailed design of FUSE, one of the most popular user-space file system frameworks. We then conducted a broad performance characterization of FUSE and we presented an in-depth analysis of FUSE's performance and resource utilization 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 when optimized, FUSE degrades their performance by up to 83%. Finally, in terms of CPU utilization, the increase in the consumption of CPU cycles by FUSE is in the range of 3× to 18× more compared to Ext4; and the relative CPU utilization ranges from only 0.2% to 31% more compared to Ext4.

Future Work. There is more room for improvement in FUSE performance, especially for metadata operations and small reads/writes. We plan to add support for compound FUSE requests and investigate the possibility of shared memory between kernel and user spaces for faster communications.

ACKNOWLEDGMENTS

We thank the anonymous FAST 2017 reviewers and our shepherd Tudor Marian for their valuable comments on that paper. We also would like to thank several people from the FUSE developers mailing list for their help understanding FUSE's behavior, especially FUSE's current maintainers, Miklos Szeredi and Nikolaus Rath, for their valuable corrections on our FAST Paper.

REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. 1986. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Technical Conference*. USENIX Association, 93–112.

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

[3] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. 1997. Extending the operating system at the user level: The ufo global file system. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, 77–90.

[4] Apache Foundation. 2010. Hadoop. http://hadoop.apache.org.

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

[6] Avfs. 2019. AVFS: A Virtual Filesystem. http://avf.sourceforge.net/.

[7] Jens Axboe. 2009. Per Backing Device Write Back. 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. 2009. *PLFS: A Checkpoint Filesystem for Parallel Applications*. Technical Report LA-UR 09-02117. LANL. http://institute.lanl.gov/plfs/.

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

[10] Claudio Calvelli. 2019. PerlFS. http://perlfs.sourceforge.net/.

[11] Zhen Cao, Vasily Tarasov, Hari Raman, Dean Hildebrand, and Erez Zadok. 2017. On the performance variation in modern storage stacks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 329–343.

[12] R. Card, T. Ts'o, and S. Tweedie. 1994. Design and implementation of the second extended filesystem. In *Proceedings to the 1st Dutch International Symposium on Linux*.

[13] Mark Carlson, Alan Yoder, Leah Schoeb, Don Deel, Carlos Pratt, Chris Lionetti, and Doug Voigt. 2015. SNIA - Advanced Storage and Information Technology: Software Defined Storage. http://datastorageasean.com/sites/default/files/snia_software_defined_storage_white_paper_v1.pdf.

[14] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Subramony, and Erez Zadok. 2017. vNFS: Maximizing NFS performance with compounds and vectorized I/O. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 301–314.

[15] Michael Condict, Don Bolinger, Dave Mitchell, and Eamonn McManus. 1994. Microkernel modularity with integrated kernel performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI'94)*.

[16] Jonathan Corbet. 2009. In defense of per-BDI writeback. http://lwn.net/Articles/354851/.

[17] B. Cornell, P. A. Dinda, and F. E. Bustamante. 2004. Wayback: A user-level versioning file system for Linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*. USENIX Association, 19–28.

[18] Mathieu Desnoyers. 2016. Using the Linux Kernel Tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.

[19] ext4-doc. 2019. Ext4 Documentation. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[20] Filebench. 2016. Filebench. https://github.com/filebench/filebench/wiki.

[21] Jeremy Fitzhardinge. 2019. Userfs. www.goop.org/~jeremy/userfs/.

[22] fuse-appendix. 2018. FUSE Library Options and APIs. http://www.fsl.cs.stonybrook.edu/docs/fuse/fuse-article-appendices.html.

[23] FUSE BDI Max Ratio. 2019. FUSE - BDI Max Ratio Discussion. https://sourceforge.net/p/fuse/mailman/message/35192764/.

[24] FUSE Destroy message. 2019. FUSE - Destroy message discussion. https://sourceforge.net/p/fuse/mailman/message/27787354/.

[25] FUSE forgets queue. 2019. FUSE - Forgets queue. https://sourceforge.net/p/fuse/mailman/message/26659508/.

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

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

[28] V. Henson, A. Ven, A. Gud, and Z. Brown. 2006. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the 2nd Workshop on Hot Topics in System Dependability (HotDep'06).* ACM SIGOPS.

[29] Hurd. 2019. GNU Hurd. www.gnu.org/software/hurd/hurd.html.

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

[31] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. 2006. Operating system profiling via latency analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06).* ACM SIGOPS, 89–102.

[32] lessfs. 2012. Lessfs. www.lessfs.com.

[33] Linus Fuse. 2019. Linus Torvalds doesn't understand user-space filesystems. http://redhatstorage.redhat.com/2011/06/28/linus-torvalds-doesnt-understand-user-space-storage/.

[34] macfuse. 2013. FUSE for macOS. https://osxfuse.github.io/.

[35] Pavel Machek. 2019. PODFUK—POrtable Dodgy Filesystems in Userland (hacK). http://atrey.karlin.mff.cuni.cz/machek/podfuk/podfuk-old.html.

[36] Pavel Machek. 2019. UserVFS. http://sourceforge.net/projects/uservfs/.

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

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

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

[40] NetBSD Foundation. 2013. The Anykernel and Rump Kernels. http://wiki.netbsd.org/rumpkernel/.

[41] nimble_casl. 2019. Nimble's Hybrid Storage Architecture. http://info.nimblestorage.com/rs/nimblestorage/images/nimblestorage_technology_overview.pdf.

[42] ntfs3g. 2019. NTFS-3G. www.tuxera.com.

[43] David Pease, Arnon Amir, Lucas Villa Real, Brian Biskeborn, Michael Richmond, and Atsushi Abe. 2010. The linear tape file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10).*

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

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

[46] RedHat's GlusterFS. 2019. GlusterFS. http://www.gluster.org/.

[47] F. Schmuck and R. Haskin. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02).* USENIX Association, 231–244.

[48] sdfs. 2012. Opendedup. www.opendedup.org.

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

[50] Swaminathan Sundararaman, Laxman Visampalli, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2011. Refuse to crash with Re-FUSE. In *Proceedings of the 6th Conference on Computer Systems.* ACM, 77–90.

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

[52] M. Szeredi. 2005. Filesystem in Userspace. (February 2005). http://fuse.sourceforge.net.

[53] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. 2015. Terra incognita: On the practicality of user-space file systems. In *HotStorage'15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage.* USENIX.

[54] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *The USENIX Magazine* 41, 1 (March 2016), 6–12.

[55] The Linux Kernel Page Cache. 2019. The Linux Kernel Page Cache. http://sylab-srv.cs.fiu.edu/lib/exe/fetch.php?media=paperclub:lkd3ch16.pdf.

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

[57] Linux Torwalds. 2019. Re: [PATCH 0/7] overlay filesystem: request for inclusion. https://lkml.org/lkml/2011/6/9/462.

[58] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. 2010. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of the FAST Conference.*

[59] Sage Weil. 2019. Linus vs FUSE. http://ceph.com/dev-notes/linus-vs-fuse/.

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

[61] E. Zadok and J. Nieh. 2000. FiST: A language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, 55–70.
[62] zfslinux. 2016. ZFS for Linux. www.zfs-fuse.net.
[63] B. Zhu, K. Li, and H. Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*.

Received January 2018; revised December 2018; accepted January 2019