DEFUSE: An Interface for Fast and Correct User Space File System Access

JAMES LEMBKE, Purdue University, USA and Milwaukee School of Engineering, USA
PIERRE-LOUIS ROMAN, Università della Svizzera italiana (USI), Switzerland
PATRICK EUGSTER, Università della Svizzera italiana (USI), Switzerland, Purdue University, USA, and TU Darmstadt, Germany

Traditionally, the only option for developers was to implement file systems (FSs) via drivers within the operating system kernel. However, there exists a growing number of file systems (FSs), notably distributed FSs for the cloud, whose interfaces are implemented solely in user space to (i) isolate FS logic, (ii) take advantage of user space libraries, and/or (iii) for rapid FS prototyping. Common interfaces for implementing FSs in user space exist, but they do not guarantee POSIX compliance in all cases, or suffer from considerable performance penalties due to high amounts of wait context switches between kernel and user space processes.

We propose DEFUSE: an interface for user space FSs that provides fast accesses while ensuring access correctness and requiring no modifications to applications. DEFUSE achieves significant performance improvements over existing user space FS interfaces thanks to its novel design that drastically reduces the number of wait context switches for FS accesses. Additionally, to ensure access correctness, DEFUSE maintains POSIX compliance for FS accesses thanks to three novel concepts of *bypassed file descriptor (FD) lookup*, *FD stashing*, and *user space paging*. Our evaluation spanning a variety of workloads shows that by reducing the number of wait context switches per workload from as many as 16,000 or 41,000 with FUSE down to 9 on average, DEFUSE increases performance 2× over existing interfaces for typical workloads and by as many as 10× in certain instances.

CCS Concepts: • **Software and its engineering → File systems management**; **Software performance**; **Consistency**.

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

ACM Reference Format:
James Lembke, Pierre-Louis Roman, and Patrick Eugster. 2022. DEFUSE: An Interface for Fast and Correct User Space File System Access . *ACM Trans. Storage* 18, 3, Article 22 (August 2022), 29 pages. https://doi.org/10.1145/3494556

1 INTRODUCTION

File systems (FSs) provide a common *interface* for applications to access data. These interfaces provide an abstract, high-level representation of a *file*, and the FS driver provides the mechanism to translate this abstraction into input/output (I/O) operations sent to physical storage media.

Work funded in parts by ERC grant #617805, NSF grant #1618923, and SNSF grant #200021_197353.
Authors' addresses: James Lembke, Purdue University, 610 Purdue Mall, West Lafayette, IN, 47907, USA , Milwaukee School of Engineering, 1025 N Broadway, Milwaukee, WI, 53202, USA, lembkej@purdue.edu; Pierre-Louis Roman, Università della Svizzera italiana (USI), Via Giuseppe Buffi 13, 6900, Lugano, Switzerland, romanp@usi.ch; Patrick Eugster, Università della Svizzera italiana (USI), Via Giuseppe Buffi 13, 6900, Lugano, Switzerland , Purdue University, 610 Purdue Mall, West Lafayette, IN, 47907, USA , TU Darmstadt, Karolinenpl. 5, 64289, Darmstadt, Germany, eugstp@usi.ch.

1553-3077/2022/8-ART22 $15.00
https://doi.org/10.1145/3494556

1.1 Implementing new FSs

Traditionally file systems (FSs) are implemented within the operating system (OS) *kernel* [27, 54, 95] and accessed via system calls defined in the interface. For new FSs, implementing drivers fitting kernel-space interfaces requires effort; porting libraries that exist only in user space [18, 41, 49] may even be required. Despite this development effort, kernel-space implementations may be short-lived and removed once deemed obsolete [73].

User space FSs can provide significant benefits over kernel space implementation methods traditionally available to developers. E.g., in the event of a crash or security breach, having the FS driver isolated from the rest of the kernel reduces the risk of data corruption [19, 45]. In particular, microkernel-based OSs extend the rationale behind user space isolation to most services and as such employ user space FSs by default and to great avail [26, 82]. User space FSs can also enable rapid FS prototyping in industrial and academic research [15, 38].

Additionally, distributed storage systems such as cloud FSs (e.g., GlusterFS [33], Databricks File System [23], Alluxio [46], Hadoop distributed file system (HDFS) [18]) and cloud object storage systems (e.g., Google Cloud Storage [34], Amazon S3 [7]) do not have kernel-space implementations, requiring applications to conform to user space application programming interfaces (APIs). Other state-of-the-art distributed FS drivers [33, 36, 37, 57, 63, 74, 81] are implemented in user space to benefit from libraries (e.g., Boost C++ library) or programming languages (e.g., Java) not available in kernel space. While command line tools for accessing some of these FSs exist [3, 9, 35], these tools are also FS-specific. Thus applications must conform to APIs of specific FSs which can be cumbersome when multiple FSs are accessed, or FSs are substituted.

1.2 Interface Requirements

A sensible solution for a user space FS interface must meet the needs in:

Flexibility: applications accessing kernel space FS implementations are not required to have knowledge of the underlying FS driver nor require modifications when using different FSs. A user space interface should not require more. This allows applications to access a diverse set of FSs without additional implementation efforts.

Efficiency: the overhead of the FS interface should be minimal to reach access speeds closest to kernel space. In particular, interfacing with user space FSs can be prone to wait context switches (waits for short), i.e., context switches used to wait for the completion of I/O operations that involve time-consuming buffer copy between user and kernel space [93].

Access Consistency: kernel-space implementations provide a set of guarantees for consistent accesses to files within the FS, e.g. locking semantics, permissions, inheritance of file descriptors (FDs) between parent and child processes. In order to establish trust in the FS driver, any user space implementation must also provide the same set of guarantees. In the remainder of this paper, we refer to these guarantees on "access consistency" simply as "consistency" for simplicity.

Existing generic solutions fall short on at least one of these requirements. That is, there is no interface for user space FSs that provides flexibility, consistency, and efficiency *all together*.

1.3 State of the Art

State-of-the-art solutions rely on one of the following methods to provide user space FS interfaces: an FS-specific user space library, an `LD_PRELOAD`-loaded library, or FUSE.

User space FS libraries provide an interface for FSs through compile-time binding to the drivers' APIs. This method is neither flexible, as each API is unique and requires compile-time binding, nor consistent, as user space FS libraries do not conform to any standard for consistent file behavior.

The LD_PRELOAD [96] method relies on diverting system calls typically wrapped by a library (e.g., libc [51]) by pre-loading another library with the same interface with the goal of redefining system call wrappers. This method can suffer from incorrect behavior that lead to failures in distributed environments (e.g., data loss, cascading errors, outages [14]), especially with files whose FD are inherited (e.g., Spark [99] passes data via the FS between manager and forked workers). Due to the distributed nature of remote execution, such incorrect behavior can be difficult to detect.

Filesystem in user space (FUSE) [48] provides a common interface for an FS driver to be implemented by a user space server while still allowing applications to access FSs using kernel-space system calls. While FUSE can be used to access some of the cloud FSs mentioned above [2, 6, 8, 30, 58, 67], FS accesses performed with FUSE suffer from major slowdowns inherent to its design [93].

1.4 DEFUSE

We present DEFUSE, a novel interface for user space FSs that employs, and combines the benefits of, a kernel-space FS driver and a user-space library. DEFUSE offers significant performance improvements over FUSE while maintaining FD consistency unlike in aforementioned approaches [4, 59, 64, 71, 75, 85, 88, 96, 100], in particular when FDs are shared between parent and child processes.

DEFUSE achieves these characteristics through three novel concepts:

(1) *bypassed FD lookup* reduces the number of wait context switches and ensures that FDs are managed by the kernel thus improving performance and achieving correct FDs behavior;
(2) *FD stashing* ensures continued correct behavior of FDs after the address space of a process is cleared following a common fork/exec;
(3) *user space paging* further ensures correct behavior of memory-mapped files. Our experiments show DEFUSE provides up to 10× the performance of the state of the art (i.e., FUSE) for user space FSs.

1.5 Contributions and Outlook

This paper makes the following contributions. After reviewing the requirements and methods for interfacing FSs and their pros and cons in Section 2, we:

- outline some of the main challenges of creating a flexible, consistent, and efficient user space FS interface;
- present the design of DEFUSE, centering on its novel concepts of bypassed FD lookup, FD stashing, and user space paging, and the features the design enables, such as POSIX compliance for FS accesses and fault tolerance;
- discuss DEFUSE's implementation, examples of user space FS integration with DEFUSE, and practical deployment of user space FSs with DEFUSE;
- evaluate DEFUSE's performance on several workloads and FSs, including synthetic workloads generated and benchmarked with IOzone [39], Linux kernel compilation,and distributed machine learning using Spark [10] involving network communication. We evaluate DEFUSE against direct kernel mount, LD_PRELOAD-loaded library, and FUSE. Results show DEFUSE outperforms FUSE by increasing throughput 2× on average for distributed workloads and as high as 10× in certain cases. We show that this increased throughput directly results from having less waits in DEFUSE.

The remainder of the paper is structured as follows. Section 3 details the challenges for creating user space FS interfaces. Section 4 presents the design of DEFUSE, and Section 5 its implementation. We evaluate DEFUSE's performance in Section 6. Finally we compare DEFUSE with related work in Section 7, and draw conclusions in Section 8.

Table 1. Summary of pros and cons of user space FS interfaces.

FS interface	Flexibility	Efficiency	Consistency
User space FS library	×	✓	×
`LD_PRELOAD` library	✓	✓	×
FUSE	✓	×	✓
DEFUSE	✓	✓	✓

Our implementation and test environment scripts are freely available in our code repository.[1]

2 MOTIVATION

In this section, we discuss in more detail the requirements needed for practical user space FS interfaces and pinpoint how the main existing approaches fail to satisfy these requirements.

2.1 User Space FS Interface Requirements

Flexibility – FS-agnostic accesses. FSs implemented as a user space library require applications to bind to the library's API at compile-time in order to access the FS. Through calls to the API, the FS library performs the necessary I/O operation; it typically involves a system call to a kernel space driver, that backs the FS, which result is returned to the application. Execution remains within user space except for system calls performed by the FS library. Command line tools such as `cp` or `find` may require access to multiple FSs through a common interface and cannot be easily re-compiled with each user space file implementation. As a result FS interfaces must be standardized.

Efficiency – low access overhead. To ensure that as much system resources as possible are allocated to useful computations, the overhead of the OS interface should remain as low as possible. FS access is no exception to this hence any user space FS interface should provide minimal overhead to FS accesses with speeds matching those of a kernel-space FS implementation.

Conceptually, the time to perform an I/O system call can be divided into two parts: (i) the time to service the system call (either through the kernel, the user space library, and/or the FUSE server) and (ii) the time to perform the I/O operation to the storage media. The time to service the system call can be considered overhead for the I/O operation as the useful work of completing the I/O is reading/writing data to media. In some cases the overhead can be dwarfed by long I/O times to storage media. However, as the speed of storage media continues to improve (e.g., solid state drives, non-volatile memory), the system call service time, and thus the overhead of FS interfaces, becomes more prominent. For example, the time to service the system call (e.g., trap, context switch to kernel space) takes 10 ms while the I/O operation takes 100 ms, the resulting OS overhead is only 10%. However if the I/O operation takes only 10 ms, then the OS overhead jumps to 100%.

Consistency – POSIX compliance for FS accesses. POSIX provides guarantees to applications that are consistent across OSs, including guarantees on FS accesses. These guarantees on access consistency (or just consistency) allow applications to execute without intricate knowledge of the underlying system implementation.

As part of this, thanks to its standard interface, POSIX also guarantees the consistent behavior of FDs for FSs. These include, among others, protection from file corruption when multiple threads access a file concurrently, and valid FDs inheritance from parent processes after a `fork` operation.

[1] https://github.com/jalembke/defuse

2.2 Background

We identify user space FS libraries, LD_PRELOAD-loaded (user space) libraries, and FUSE as the three main methods used to provide user space FS interfaces. Table 1 summarizes how these methods comply with the aforementioned requirements.

User space FS library. With a user space FS library, the FS driver is implemented in a code library which the application binds to at compile-time. Examples of this include the MPI-IO library [22] used by MPI, the HDFS C API `libhdfs` [47], and the Amazon Web Services (AWS) SDK for C++ [13].

Flexibility (×): A user space FS library does not provide the flexibility of a kernel space system call. Since there is no standardized API for user space FSs, applications must have prior knowledge of FS function calls to perform I/O operations. Access to multiple FSs implemented as user space FS libraries requires conforming to each FS API. For example, `libhdfs` uses `hdfsOpenFile` to open a file and `hdfsCloseFile` to close it. AWS SDK uses different calls.

Efficiency (✓): Due to the direct binding to the user space FS library, calls to the FS tend to be efficient. Access to FS operations are made through direct function calls.

Consistency (×): User space FS libraries do not necessarily provide consistency guarantees. Since the implementation uses its own API for FS access and file metadata is maintained within the library, it is not required to comply to a standardized interface as with FUSE or kernel-space (enforced through the virtual file system (VFS)). Thus POSIX behavior is not guaranteed [28].

LD_PRELOAD *library.* A user space FS library does not conform to a standard API such as POSIX. This limitation can be prohibitive to applications that desire to use a user space library, but are created to operate using POSIX calls. Therefore, the research community has explored several methods to implement POSIX-looking efficient interfaces for user space FSs using LD_PRELOAD [4, 59, 64, 71, 75, 88, 96] or linkable libraries [85, 100] that provide access to FSs by intercepting system calls. An LD_PRELOAD (-loaded) library, being loaded before `libc`, causes its versions of the functions to be invoked in lieu of those of `libc`. Intercepted I/O calls are sent to the user space FS library via its API, keeping control flow within user space until the operation must be passed to the kernel. An application thus executes as if calls were made to the kernel. Accessing different FSs requires changing the LD_PRELOAD library.

Static binding with LD_PRELOAD. Applications that use static binding link their function calls to the function definitions at compile time rather than at runtime. Since LD_PRELOAD is based on intercepting function calls at runtime by being loaded before `libc`, it cannot intercept applications that use static binding.

Flexibility (✓): Similarly to FUSE and kernel space implementations, applications access FSs via system calls. Applications only need knowledge of existing system calls for FS access, not of the underlying FS driver. This flexibility is provided when multiple FSs are accessed, granted each FS has its own LD_PRELOAD library to intercept those calls.

Efficiency (✓): As system calls are intercepted and redirected in user space, the efficiency of the LD_PRELOAD approach is comparable to that of a user space FS library.

Consistency (×): The POSIX behavior of FDs and return codes is not guaranteed with the LD_PRELOAD scheme, especially for FDs inherited from parent to child processes (cf. Section 3.2.1). Furthermore, the LD_PRELOAD library must invoke the user space FS library's API which does not necessarily comply with a standard for file metadata representation. Thus, to ensure POSIX behavior, LD_PRELOAD must maintain a mapping between the FS library's file metadata and FDs returned to the application. Maintenance of this mapping is trivial in most cases and can consist of a hash translating an FD, manufactured by the LD_PRELOAD library (e.g., a number) when it intercepts an open library wrapper call, to the metadata of the FS library. This table

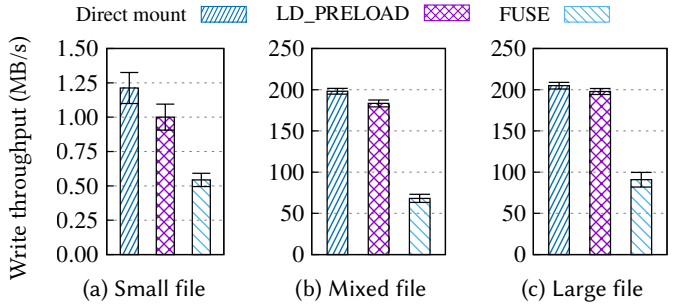

Fig. 1. Write throughput of a direct mounted `ext4` FS, an `LD_PRELOAD` library and a `FUSE` wrapper around `ext4` (higher is better). FUSE's throughput ranges between 34% and 44% that of direct mount.

can be stored in the application's memory; if the table is lost, as with a call to `exec`, FDs are invalid in a child process.

Filesystem in user space (FUSE). With FUSE, when a system call induces an I/O operation sent to the kernel, the FUSE kernel driver sends the I/O request to user space through a FUSE virtual device (`/dev/fuse`). The request is received by the FUSE server in user space and sent to the relevant FS library through an interface similar to that of the VFS [53]. The I/O operation is then processed by the user space server (and may include calls to kernel resources). Results are passed back through the FUSE device to the kernel driver, and finally to the application that issued the I/O operation.

Flexibility (✓): FUSE provides the benefit of FS-agnostic access for applications through system calls without prior knowledge of the underlying FS.

Efficiency (×): The transfer of control between kernel and user space when servicing a system call — a core design decision behind FUSE — requires costly context switching and buffer copying between processes of different spaces. This leads to serious performance penalties (cf. Section 3.1) which hinders deployments in production environments, centering FUSE's use to FS prototyping [1].

Consistency (✓): While I/O requests are processed by the FUSE server in user space, as the application accesses the FS through system calls, POSIX compliance is enforced by FUSE's kernel driver. FUSE manages a mapping in user space from `inode` numbers maintained by the kernel to FUSE file information passed to the FS implementation.

3 CHALLENGES

This section elaborates on challenges in achieving a user space FS interface that reconciles direct mount's efficiency with FUSE's flexibility and consistency.

3.1 Efficiency

While FUSE is both flexible and consistent, it suffers from poor performance due to a high number of waits, i.e., voluntary context switches used to wait for the completion of an I/O operation [52]. We demonstrate this overhead in Figure 1 and Figure 2 that depict write throughput and number of waits, respectively. We compared three interfaces: (i) direct access to an `ext4` FS using POSIX system calls to the kernel-space `ext4` driver, (ii) an `LD_PRELOAD` library storing its own FD table, and (iii) a FUSE server that simply wraps over POSIX calls (`fusexmp` [29]). Interfaces were tested with three write-based workloads: (a) small file – 4,096 writes of 128 B to different files, (b) mixed

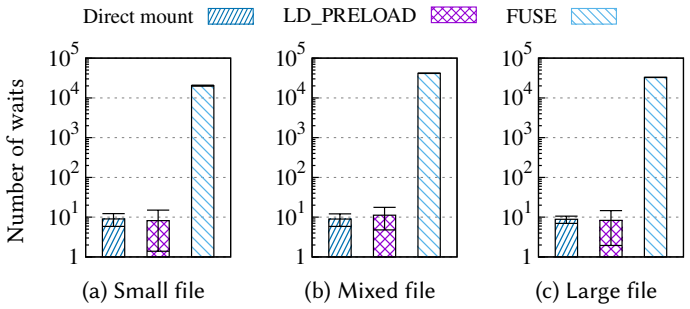

Fig. 2. Number of waits for the benchmarks used in Figure 1 obtained with the command `time %w` [52] (log scale, smaller is better). FUSE causes tens of thousands of waits while direct mount and LD_PRELOAD only ≈9.

file – 2,048 writes of size linearly distributed between 64 B and 128 kB to different files, and (c) large file – 1 write of 128 MB to a single file.

FUSE's throughput, shown in Figure 1, is visibly worse than the kernel-space baseline in all workloads: 43% and 44% of the direct mount throughput for the small file and large file ones, respectively, and even 34% for the mixed file one. The LD_PRELOAD library performs much better, especially when there are fewer operations since LD_PRELOAD's overhead is proportional to the number of system calls intercepted, with throughputs of 87%, 93%, and 97% of direct mount for small, mixed, and large file workloads, respectively. Figure 2 further shows that FUSE's throughput drop is mostly due to the considerable amount of waits, with numbers as high as ≈20,000, ≈41,000, and ≈33,000 for small, mixed, and large file workloads, respectively. The LD_PRELOAD library however is on par with the direct mount solution, causing less than 10 context switches on average.

Specifically, an I/O wait entails a context switch between kernel and user space FUSE server which includes (1) saving processor state, (2) changing processor mode, and (3) copying data between kernel and user space buffers [93]. While (1) and (2) induce little overhead, (3) is costly.

3.2 Consistency

The LD_PRELOAD library interface seems attractive in light of the results presented in Section 3.1. However, this interface as well as user space FS libraries are subject to two challenges related to FS consistency: the first concerns FD heritage and the second memory-mapped (user space) files.

3.2.1 FD heritage dilemma. As explained in Section 2.2, when an LD_PRELOAD library is used, the mapping table between FDs and user space file metadata may be stored in memory. However, while open FDs and the internal state of the FS maintained by the LD_PRELOAD library are copied from parent to child upon a `fork`, the FS state is destroyed when the child invokes `exec` if it is in memory.

Illustration. Consider the processes in Figure 3: (a) User process p, in the LD_PRELOAD environment, executes library function wrapped system calls that are intercepted by the LD_PRELOAD library. (b) Both the LD_PRELOAD and FS libraries are stored in shared memory and the internal state of the FS resides within the address space of p. (c) When p performs an `open` system call, the LD_PRELOAD library makes the corresponding call to the FS library. The FS returns an internal representation of the file as F. Process p expects an FD number to be returned from `open`, not the internal representation of F. (d) To avoid passing the `open` call to the kernel (to not increase latency), the LD_PRELOAD library manufactures FD 4, updates its internal file mapping 4 to F, and returns the manufactured FD to p. This manufactured FD is only valid in the context of the LD_PRELOAD library and only if the mapping is in memory. Future system calls using FD 4 will be intercepted by the

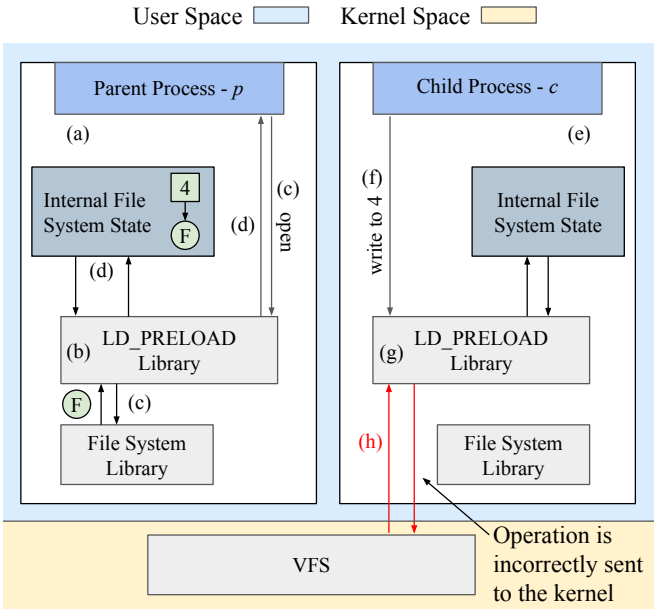

Fig. 3. Illustration of an error caused by FDs inherited between parent and child with the LD_PRELOAD library.

LD_PRELOAD library, mapped to F and passed to the FS library. (e) Process p creates child process c which inherits, among other things, the open FDs and the memory of the LD_PRELOAD library. After performing an exec, the child's address space, including the internal state for manufactured FDs, is destroyed by the executable of c. (f) Process c later writes to the inherited FD 4. (g) As the LD_PRELOAD library memory was cleared during exec, the library cannot map FD 4 to F, so it passes this I/O to the kernel. (h) The kernel does not know what FD 4 references since the LD_PRELOAD library of parent process p manufactured it. Ultimately, the operation fails in the kernel with an invalid FD and is returned to c via the LD_PRELOAD library.

Prominent examples. The FD heritage dilemma occurs commonly in shell file redirection. Consider a shell running under the context of an LD_PRELOAD library, where the user executes a cloud application running in Spark [99] by running `spark-submit ... > out.txt` to capture the output to a file located in a user space FS. The shell opens `out.txt` which is intercepted by the LD_PRELOAD library prior to creating the `spark-submit` process. Due to the FD heritage dilemma, the FD for the output file `out.txt` inherited by the child is no longer valid after the exec call to invoke `spark-submit`. The FD heritage dilemma is also common in widely used tools such as gcc that create child processes for performing subtasks, and in resource manager frameworks such as the PySpark workload daemon [76], which creates child processes for worker tasks at runtime, with each child task sending and receiving data to and from the daemon through FDs.

Impact in other interfaces. The user space FS library interface suffers from a related problem if a parent process needs to pass file metadata to a child. The application must be aware of the FS library API and does not access the FS using FDs, it instead uses the user space file metadata directly. As with open FDs, child processes cannot inherit file metadata.

FUSE is not affected by the FD heritage dilemma since the FUSE server is a single process, separated from the application processes. Hence the FUSE identifier table is unique. When a parent opens a file, the corresponding (inheritable) FD matches a unique `inode` within kernel space.

Similarly to FUSE, in-kernel FSs are not affected by FD heritage dilemma since all file structures are managed within the kernel address space. References to files are synchronized by the FS driver within the kernel to ensure consistency when accessed by applications via FDs. The FDs maintained by the kernel and inherited by child processes map to a unique `inode` within kernel space.

3.2.2 Memory-mapped files. Mapping files into memory using `mmap` is a common method to share memory between processes. The `LD_PRELOAD` interface cannot provide consistent memory-mapping of files hosted by user space FSs for the two following reasons.

First, while an `LD_PRELOAD` library can intercept explicitly invoked system calls such as `mmap`, it cannot intercept implicit I/O operations. For instance, accesses to a memory-mapped file such as reads and writes are performed using `load` and `store` paging instructions to the address region of the mapped memory. These instructions, which may result in an I/O operation to the underlying file, are not explicit system calls and therefore cannot be intercepted.

Second, as a direct consequence of the FD heritage dilemma, a child process cannot access a user space file mapped in memory if the mapping has been done by its parent process. For example, consider the following set of actions taken by a process: (a) A process opens a user space file, the `open` call is intercepted and redirected to the user space FS library. (b) The process then maps this file into its address space using `mmap` for shared access, however since the FD is manufactured by the intercepting FS library, the call to `mmap` fails because the FD is invalid.

Impact in other interfaces. Directly linked user space FS libraries equally suffer from the same problem for `mmap`-ed files as `LD_PRELOAD` libraries do. FUSE does not, however, since the FUSE kernel driver translates paging operations to `read` or `write` sent to the user space FUSE server.

4 DEFUSE DESIGN

We present the design of DEFUSE centering on its novel concepts and the features it enables.

4.1 Overview

DEFUSE uses a unique four-fold approach to be the first solution that achieves flexibility, efficiency, and consistency all together (cf. Table 1). As shown in Figure 4, our approach comprises:

(1) a hook into `libc` to intercept and forward FS access calls to a user space FS library — for flexibility and efficiency (Section 4.2);
(2) an FS kernel driver providing *bypassed FD lookup* semantics to ensure FDs are managed by the kernel and remain correct when inherited by child processes — for consistency and efficiency (Section 4.3);
(3) a shared memory space for *FD stashing*, allowing FS metadata to be restored after a process's address space is cleared by an `exec` call — for consistency (Section 4.4);
(4) a memory management framework for user space paging to correctly handle memory mapped pages backed by user space files — for consistency and efficiency (Section 4.5).

We demonstrate how DEFUSE's design maintains POSIX compliance for FS accesses (Section 4.6), why it provides better fault tolerance than FUSE (Section 4.7) and how it delegates caching to the respective user space FS libraries to ensure cache consistency (Section 4.8).

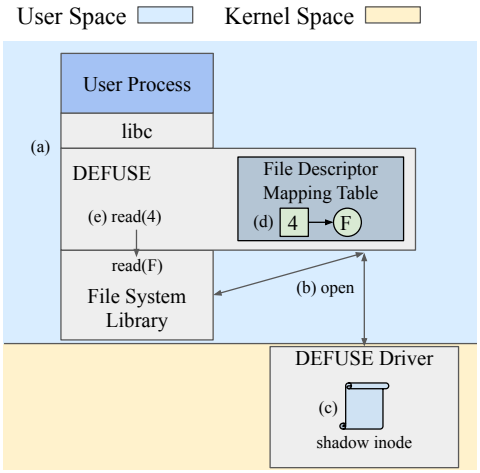

Fig. 4. Handling of I/O requests in DEFUSE. I/O requests are intercepted by DEFUSE's `libc` extension and directed to the user space FS library. Requests for open are sent to the DEFUSE kernel driver to allocate a valid `inode` and FD. FDs are temporarily stashed when an `exec` is intercepted (cf. Figure 5).

4.2 System Call Redirection

Performance gains of DEFUSE come from *redirecting I/O operations directly to the user space FS library* through hooks in `libc`, thus avoiding waits presented in Section 3.1. Such hooks are used in other systems for language extension [69] or behavior customization [25]. DEFUSE implements these hooks by directly modifying the system call wrapper functions within the `libc`. Once installed, applications that dynamically link to `libc` are automatically be able to take advantage of the DEFUSE functionality, while those that statically link against `libc` needs to be recompiled. While implementing system call redirection could have also been implemented using an `LD_PRELOAD` library, from our experience, the environmental setup of `LD_PRELOAD` can be problematic.

In addition to wrapped system calls, applications may also execute direct system calls that are made through direct invocation of a trap. These direct calls cannot be intercepted neither by hooks in `libc` nor using `LD_PRELOAD`, and hence fall out of the scope of DEFUSE. However, invoking direct system calls can be a cumbersome task for application developers [80] as it requires knowledge of the application binary interface (ABI) used for system calls. Since the ABI may differ between OSs and architectures, using the ABI directly can result in portability issues. As such, it is our understanding that most applications rely on `libc` wrappers for system call invocation for convenience and can use DEFUSE without strong limitations in their functionalities.

4.3 Bypassed FD Lookup

Every `open` system call received by the kernel requires the `inode` of the file to be resolved through one or more `lookup` requests. One `lookup` request is required per level of the directory tree to ensure the file path is valid and the user has sufficient access permission. When the file being opened is hosted on an FS accessed via FUSE, this requires a separate request sent by the FUSE kernel driver to the user space FS library and thus require waits. Opening a file deep in the directory tree with FUSE can be costly because of this series of `lookups`.

To avoid the many transfers between kernel and user space, DEFUSE makes use of its own kernel driver to provide *bypassed FD lookup*. Thanks to bypassed FD lookup, the kernel does not need to

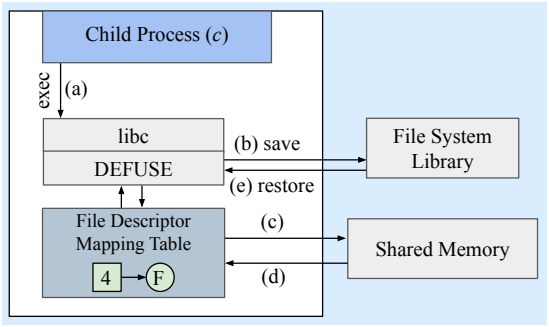

Fig. 5. Depiction of how DEFUSE's FD stashing uses a shared memory segment to save and restore the internal state of the DEFUSE FD mapping table before and after the invocation of an exec system call.

send the lookup requests to a server in user space when it opens a file. Instead of resolving to a real inode for the file, the DEFUSE kernel driver creates and splices a shadow inode (i.e., add an entry [86]) in the directory cache of the FS, then allocates an FD for the file, and finally allows the open to proceed to the user space FS library. This shadow inode represents a placeholder for the file within the directory cache of the system and is not usable for read and write operations. In turn, file permission checks, reading, and writing are performed by the user space FS library and enforced by the DEFUSE libc extension (cf. Section 4.2). The shadow inode is removed by the kernel when all references to its file are removed. This may happen through explicit close system calls related to the file and/or upon termination of the process, either explicitly or implicitly (e.g., process crash), as the kernel frees all resources allocated to the process.

Figure 4 depicts the management of I/O requests, and in particular open requests, by DEFUSE:

(a) An I/O system call is generated by the application. The system call is invoked as a library call to libc, which is intercepted by DEFUSE as explained in Section 4.2.
(b) For open system calls, the operation is sent to the FS library to update the internal state of the FS as well as retrieve a reference to the FS metadata. The open call is also sent to the kernel to allocate a valid FD.
(c) The DEFUSE kernel driver allocates a shadow inode, initializes the FD entry for the file, and returns the FD to the user application. Once an FD is created by the DEFUSE kernel driver, it serves as index of the FD mapping table managed by DEFUSE to retrieve the matching user space FS metadata. The FD mapping table is stored in the application process memory (cf. Figure 4 for DEFUSE's overview); hence opened files cannot be accessed by other processes.
(d) DEFUSE then maps the returned FD to the FS metadata returned from the FS library.
(e) When the application requests access to the file (e.g., read), the request is again sent to libc and intercepted by DEFUSE. DEFUSE then uses the FD mapping table to retrieve the user space FS metadata and sends the request directly to the user space FS library through its API.

Note that DEFUSE uses a shadow inode for bypassed FD lookup, which requires the system to be installed with the DEFUSE kernel module. One possible alternative involves using an existing file (e.g., /dev/null) instead of a shadow inode. This has the advantage of not requiring the DEFUSE kernel module, however, would not maintain expected FS semantics (e.g., file position pointers, fcntl locking, file data in /proc). DEFUSE complies with POSIX for FS accesses (cf. Section 4.6).

4.4 Managing FDs across exec with FD Stashing

As described in Section 3.2.1, once a `fork` system call completes, the memory of the child process is a duplicate of its parent. Subsequently, if the child process executes an `exec` system call, its memory is replaced with the memory of the program to be executed. While bypassed FD lookup ensures that FDs are correctly inherited by a child process, it does not prevent FDs from being erased alongside the rest of the process memory upon `exec`'s execution. Without additional management, future accesses by the child process to its FDs will fail.

To ensure correct FD semantics after a call to `exec`, DEFUSE utilizes a novel concept we call *FD stashing* that temporarily saves, then restores, FDs during the execution of an `exec` system call. As depicted in Figure 5, FD stashing operates as follows:

(a) The `exec` system call is intercepted by the DEFUSE `libc` extension.
(b) The `save` routine in the user space FS library is invoked to allow the FS library to perform any necessary action to save file meta data.
(c) The internal state of the DEFUSE FD mapping table is saved to a shared memory segment.
(d) After the `exec` system call, and the memory of the process erased, if the process executes an I/O system call, the FD mapping table is restored from the shared memory segment.
(e) Further, the `restore` routine in the FS library is invoked to allow the user space library to restore the saved state of the file.

Subsequent I/O calls continue to be intercepted and sent directly to the user space FS library.

While a shared memory segment is typically used to share data between multiple processes, DEFUSE instead uses a segment to share data (i.e., the FDs) between a process and a future version of itself, once the call to `exec` completes. Since the shared memory segment storing the FDs is unmapped from a process's address space after a call to `exec`, DEFUSE uses the process identifier to remap the segment when the child process resumes execution upon completion of the `exec`.

4.5 Managing `mmap` and User Space Paging

As described in Section 3.2.2, memory-mapped files are accessed via I/O operations that rely on the `load` and `store` instructions instead of system calls like `read` and `write`. These operations cannot be intercepted by the DEFUSE `libc` extension as presented in Section 4.2.

However, the `userfaultfd` interface for managing page faults in user space [87] has been added in Linux kernel 4.3. Originally implemented to help Linux-based hypervisors handle virtual machine migration, it allows a user space application to register a virtual address range with the kernel and subsequently be sent an event when a page within that range needs to be copied into memory. Events are received by reading from a special FD created by `userfaultfd` via a system call. Upon receiving an event, page data is copied into memory using the `ioctl` system call. To inform the page fault manager on virtual address space layout changes, `userfaultfd` also supports calls to `fork` and events for memory remapping (`mremap`), unmapping (`munmap`), removal (as result of a `madvise` call using `MADV_REMOVE` and/or `MADV_DONTNEED`).

DEFUSE takes advantage of `userfaultfd` to intercept and manage memory-mapped file I/O. As depicted in Figure 6, DEFUSE's user space paging operates as follows:

(a) The `mmap` system call is intercepted by the DEFUSE `libc` extension.
(b) The mapped address and corresponding FD are stored in a memory region mapping table.
(c) A `userfaultfd` is created to allow the kernel to signal to DEFUSE when a page fault within the address space occurs.
(d) When DEFUSE receives such a signal from the kernel, DEFUSE uses the memory region mapping table to determine which virtual address ranges (for which a fault was emitted) correspond to which FDs along with the offset within the file.

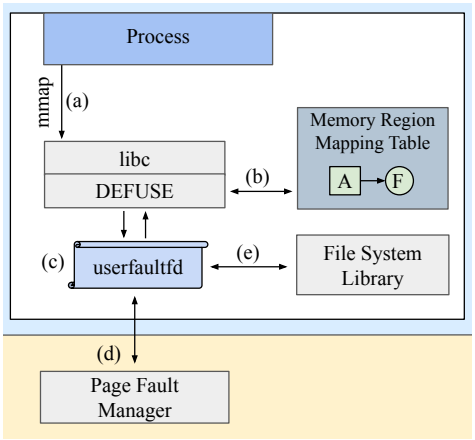

Fig. 6. Depiction of how DEFUSE utilizes `userfaultfd` and a memory mapping table for user space paging.

(e) DEFUSE then sends a `read` directly to the user space FS library copying the resulting data into the virtual address space of the process using `ioctl`.

When an address range is unmapped via `munmap`, DEFUSE intercepts the call, ensures the pages are flushed to the user space file, removes the entry from the memory region mapping table, and unregisters the `userfaultfd`. DEFUSE makes use of the `userfaultfd` events for remapped addresses (remap and fork) to ensure that the address space mapping table is kept up to date as the process's virtual address map changes.

As an alternative to `userfaultfd` for user space memory management, we could also protect the memory region using `mprotect` and have DEFUSE handle the `SIGSEGV` signal sent by the kernel. However, servicing this signal has a significant performance impact [92].

4.6 Maintaining POSIX Compliance for FS Accesses

Enforcing file permissions. File permissions are used to properly isolate applications. The kernel is responsible for checking that applications have sufficient permission when opening a file so that later read/write operations succeed. DEFUSE enforces access permissions by invoking the `access` function provided by the user space FS library upon intercepting an `open` system call. If access is not allowed DEFUSE prevents the intercepted `open` system call from completing successfully.

File position pointers. Internal file position pointers, which keep track of read/write offsets, must be consistent between parent and child processes. Consider a parent process opening a file, reading some amount of data from the file (advancing the internal file position pointer), then creating a child process passing it this opened FD. The child inherits the opened FD, along with the file position pointer, and then reads from the file expecting the data to be read at the inherited position. Since DEFUSE utilizes a DEFUSE kernel driver to create a FD in the kernel, the internal file position pointer is managed by the kernel to ensure consistency between parent and child processes. As DEFUSE intercepts all read and write operations, redirecting them to the user space FS library, the file position pointer is maintained in the kernel by using the `lseek` system call.

Parallel file access. For parallel file access, read and write operations to files must be performed atomically. For example, if one thread performs a file read, DEFUSE adjusts the file position pointer using `lseek`. However, if another thread performs a write to the same file at the same time, DEFUSE also adjusts the file position pointer using `lseek`. To ensure that there is no contention for the file

data and position pointer, DEFUSE uses `fcntl` to lock the file with each I/O operation. This lock is maintained by the DEFUSE kernel driver through the shadow `inode`. DEFUSE similarly ensures atomicity of further operations capable of generating race conditions (e.g., `dup2`, `dup3`).

Paging. POSIX behavior requires that (1) `mmap` succeeds for a valid FD, and that (2) if the file is mapped for shared memory, then the mapping should also be valid within a child process created with `fork`. DEFUSE complies with this requirement thanks to user space paging and FD stashing.

Accessing multiple FSs. In a kernel space implementation, FSs are mounted in the directory tree. This mounting allows the kernel to send I/O operations to the appropriate FS driver. Similarly, the DEFUSE kernel module also presents a FS to the system to perform bypassed FD lookup. Each user space FS must be independently associated with a DEFUSE mount. DEFUSE uses this mount to determine which user space FS library to send system calls to. Details are discussed in Section 5.3.

4.7 Fault Tolerance

Failures in an FS can result in loss of access or even total loss of the data itself if it is not written to permanent storage. Being contained within the running process, DEFUSE requires no centralized server process. DEFUSE duplicates the FS logic to each process accessing the FS and a failure in the process thus does not affect other processes using DEFUSE. In contrast, FUSE relies on a single server process running in user space that may crash (due to, e.g., program error, explicit termination by a user). Upon FUSE server failure, accesses to the FUSE FS fail for all processes in the system.

4.8 Cache Delegation

DEFUSE does not manage a cache. Instead, DEFUSE delegates caching to the user space FS libraries it relies on. While FUSE uses a single server to avoid cache inconsistencies (e.g., duplicate entries) across all FSs, DEFUSE's delegation resembles an approach using one server per process.

5 DEFUSE IMPLEMENTATION AND SEMANTICS

We present DEFUSE's implementation, the user space FSs it already supports, and how to deploy it.

5.1 Code Base and Interface

The DEFUSE driver is a Linux kernel module made up of ≈300 source lines of code (SLOC) and the DEFUSE library of ≈2,000 SLOC which includes additions and modifications to `libc`. Integrating a user space FS in DEFUSE requires implementation of ≈25 functions that are similar to the high-level interface of FUSE defined in `fuse.h` [31]. We opted for an interface similar to the high-level one of FUSE, over its low-level one, since it is simpler for developers to reason about file paths instead of `inodes` and to not have to manage the mapping between `inodes` and file paths themselves. From our experience many FUSE implementations use the high-level interface [6, 8, 49, 61] while some state-of-the-art [100] also base their implementation on the high-level interface.

Prototypes for the interface functions are provided in Listing 1. Each function is required to return an error value as described by the system error numbers (i.e., `errno`). Function output is specified by output parameters (using references). FS integration only requires a shared library that exports the needed interface functions. DEFUSE loads the library (dynamically) as needed when a user space FS is accessed.

5.2 User Space FS Integrations

To test ease of implementation, we integrated several user space FSs with the DEFUSE interface. These FSs are listed in Table 2:

```
// File system operations
void init(const char* mount_point,                  // Initialize the FS mounted at specified
          const char* backend_path);                // mount point (called after FS lib is loaded)
void finalize();                                    // Un-initialize FS (before FS lib is unloaded)

// File operations
int open(const char* path, int flags,               // Open file - out_fh is the file handle
         mode_t mode, uint64_t* out_fh);            // used to later access the opened file
int close(uint64_t fh);                             // Close a file
int read(uint64_t fh, char* buf, size_t size,       // Read data from a file
         off_t offset, size_t* bytes_read);
int write(uint64_t fh, const char* buf, size_t size,// Write data to a file
          off_t offset, size_t*  bytes_written);
int fsync(uint64_t fh, int data_sync);              // Synchronize state and store of file
int ftruncate(uint64_t fh, off_t length);           // Truncate an opened file to given length
int fgetattr(uint64_t fh, struct stat* stbuf);      // Retrieve an opened file's attributes
int save(uint64_t fh);                              // Save file handle data (before FD stashing)
int restore(uint64_t fh);                           // Restore (FD stash, then) file handle data
int getattr(const char* path,                       // Retrieve file's attributes by path name
            struct stat* stbuf, int flags);
int trunc(const char* path, off_t length);          // Truncate a file by path name
int readlink(const char* path, char* buf,           // Read a symbolic link
             size_t bufsize, size_t* bytes_written);
int unlink(const char* path);                       // Remove a file by path name

// Directory operations
int mkdir(const char* path, mode_t mode);           // Create a directory
int rmdir(const char* path);                        // Remove a directory
int getdirentcount(const char* path, int* count);   // Get directory entry count
int getdirents(const char* path, DIR* dirent_buf,   // Get directory entries
               size_t bufsiz, size_t* ents_written);
int chown(const char* path, uid_t uid, gid_t, gid); // Change ownership of a file/directory
int chmod(const char* path, mode_t mode);           // Change permissions of a file/directory
int access(const char* path, int mode);             // Check permissions of a file/directory
int rename(const char* oldpath, const char* newp);  // Rename a file/directory
int utime(const char* path,                         // Update access and modification time
          const struct utimbuf* times);             // of a file/directory
int symlink(const char* target, const char* linkp); // Create symbolic link to  file/directory
```

Listing 1. DEFUSE interface similar to that of FUSE. Green text highlights functions called by DEFUSE itself.

(1) The direct wrapper FS wraps the I/O operations directly to their corresponding system call. The corresponding implementation for FUSE is the pass-through function of `fusexmp` [29].
(2) A Virtual File System (AVFS) [12] enables applications to look inside compressed files (e.g., gzip, tar, zip) without an additional decompression tool. This FS provides a user space library that contains the functions for FS operations.
(3) CRUISE [68] is a checkpoint/restart file system for high performance computing applications. This FS provides a user space library that contains the functions for FS operations.
(4) SSHFS [49] grants applications access to remote files through the secure shell network (SSH) protocol. Our implementation consists of a port from the existing FUSE implementation.
(5) HDFS [47] grants applications access to the storage component of Hadoop without using the HDFS API. Our implementation consists of a port from the existing FUSE implementation.

Using SLOC to evaluate integration effort, we conclude it takes relatively similar effort to integrate a user space FS in FUSE and DEFUSE.

5.3 Deploying a DEFUSE-backed User Space FS

The DEFUSE kernel module parses mount parameters to know which user space FS library is associated to each mounted FS. Consider the following command to access an FS with DEFUSE:

Table 2. Integration effort of example user space FSs in DEFUSE and their FUSE equivalent.

User space FS	Integration methodology	DEFUSE SLOC	FUSE SLOC
Direct wrapper	Simple wrapper	280	402
AVFS	Simple library wrapper	304	308
CRUISE	Simple library wrapper	274	503
SSHFS	FUSE port[a] (≈150 SLOC changed)	4,803	4,956
HDFS	FUSE port[b] (≈150 SLOC changed)	26,757	26,713

[a] Ported from `libfuse` SSHFS [49].
[b] Ported from `native-hdfs-fuse` [61]. This FUSE implementation does not use `libhdfs` [47] and is therefore not bound by the use of the Java virtual machine.

```
mount -t defuse -o backend=/src/dst -o library=/usr/usfs.so defuse /mnt/fs
```

In this example the DEFUSE bypassed FD lookup FS is mounted at `/mnt/fs`, the user space library to redirect calls to is located in `/usr/usfs.so` while the back-end location for this FS is `/src/dst`. The library option must be a path in the FS (outside of DEFUSE) that contains the user space FS implementation. While the back-end is required by DEFUSE, it does not need to be a path; it is intended to serve as information for the user space FS implementation to determine how to access files and directories. It may be a path (as in the example), a file (e.g., an FS image), a network host name (e.g., for a network based user space FS), etc.

When an application makes its first access to the FS, DEFUSE retrieves the user space library path from the mount point and invokes `init` (cf. Listing 1) to initialize the FS before any other calls are processed. When the application completes, the `finalize` function is invoked to allow the user space FS to clean up any created data structures.

6 EVALUATION

We evaluated DEFUSE against other FS interfaces with workloads generated following standard methods and real-life functional workloads. Overall DEFUSE's data and metadata throughput significantly outperforms that of FUSE, in some cases achieving 10× speedups, while even our evaluations with distributed workloads involving communication over the network show that DEFUSE can still achieve 2× speedups over FUSE. Further analysis shows that speedups are a direct result of reduced wait context switch overhead, in some cases from 16,000 or 41,000 with FUSE to 9 with DEFUSE, and that DEFUSE's FD stashing displays negligible runtime overhead (e.g., 8.9 µs for 1,024 FDs). In addition to throughput, DEFUSE's user space paging performs as well as FUSE's for reads and 1.8× better for writes. Lastly, we show DEFUSE also performs up to 2× better than FUSE when accessing the user space AVFS.

Every figure throughout this section depicts averaged results with error bars of one standard deviation from the geometric mean. Most results were averaged over 50 runs, except for the results in Figure 9c, Figure 11, and Figure 12 that were averaged over 20 runs due to long runtimes.

6.1 Single-machine Evaluation

We first evaluated DEFUSE on a single machine observing I/O throughput and induced number of waits using IOzone [39], runtimes of three throughput-intensive applications (i.e., kernel archive decompression, backup and compilation), as well as runtimes of file metadata operations.

6.1.1 Setup. First we describe the setup required for our experiments.

Hardware. Benchmarks were performed on a single machine running Ubuntu 18.04.1 LTS with kernel version 4.15.0-43, two Intel® Xeon® E5-2420 processors with 2.2 GHz, 24 GB of main memory with SATA-attached 256 GB SSD and 1 TB HDD running at 7200 RPM. With the exception of `tmpfs`, which uses system memory, the backing storage for all evaluated FSs use the attached HDD.

Caching policy. FS caches were cleared between each I/O operation of the benchmarks to remove any effect of caching. Files were flushed to disk (`fsync`) after each write operation.

FSs. We used `ext4` [27], JFS [42], FAT [56], and `tmpfs` [90]. Both `ext4` and JFS were chosen due to their large install base in consumer Linux as well as industry servers. While FAT is no longer a commonplace FS for PCs or industry servers, it was chosen for evaluation due to its prevalence in external media. Finally, `tmpfs` was chosen for its low I/O overhead since it resides in memory, allowing us to isolate the FS logic overhead. Note that for FSs built on actual media, the time spent for I/O operations on the media itself may overshadow the time performing FS logic.

Baselines. We compared DEFUSE to (1) an `LD_PRELOAD` library, (2) Direct-FUSE [100], (3) FUSE (v2.9.7) with and without direct I/O enabled, and (4) an FS mounted using a kernel driver.

The `LD_PRELOAD` library maintains its own FD table (cf. Section 2.2) that is thus invalid when inherited. Direct-FUSE is a user space FS interface built as an extension to the `libsysio` library [89]. Like DEFUSE, Direct-FUSE intercepts I/O system calls using hooks in `libsysio`. However, Direct-FUSE has similar drawbacks as the `LD_PRELOAD` approach, i.e., it does not address the FD heritage dilemma nor user space paging. `LD_PRELOAD` and Direct-FUSE are inconsistent solutions, they are included for comparison only. The user space FS library for DEFUSE and FUSE is a simple wrapper over POSIX system calls, as expressed in Section 5.2. Using direct I/O with FUSE makes I/O operations skip kernel FS caches.

Benchmarking tool. We used the widely adopted IOzone [39] FS benchmarking tool — a user space application that creates, writes, and reads files of varying sizes using POSIX system calls. IOzone generated three workloads described further alongside the corresponding results in Section 6.1.2.

I/O throughputs were measured in terms of data read/written per second. The depicted results were normalized as a ratio of the direct mount to simplify comparisons (otherwise the greatly varying access performance for different FSs would render some figures' y-axis hardly readable).

6.1.2 Throughput results. We evaluated file access throughput separately for small and large file as well as for a mixture of files with different sizes.

Small files. We evaluated the FSs for small file accesses where a large quantity of system calls must be serviced. IOzone created 4,096 files of 128 B in size. This workload is prevalent in many parallel computing applications [17, 21] where large quantities (sometimes in the millions) of files smaller than 64 kB are used to store checkpoint data. Due to the large quantity of system calls needed to create, write, and read that many files, we expected FUSE's performance to be greatly affected as a large quantity of system calls requires an equally large quantity of waits.

Overall, as Figure 7a and Figure 7b show, DEFUSE always achieved within 10% of the best-performer `LD_PRELOAD` while the inconsistent Direct-FUSE was within 1%. DEFUSE reached at best equivalent throughput to direct mount (write with FAT) yet at worse 50% (read with `tmpfs`). The lower performance is primarily caused by the additional time spent resolving FS metadata from the FD mapping table, however it is only minor when compared to the overhead of FUSE. Direct I/O for FUSE has negligible effect on performance as the benefits are lost due to the small buffer sizes and the large number of system calls to be serviced (cf. discussion on Figure 8).

The direct mount actual throughput varied across FSs: (a) `ext4` had 2.6 MB/s write and 6.8 MB/s read throughputs, (b) JFS was slower with 1.1 MB/s write and 4.8 MB/s read throughputs, (c) FAT was

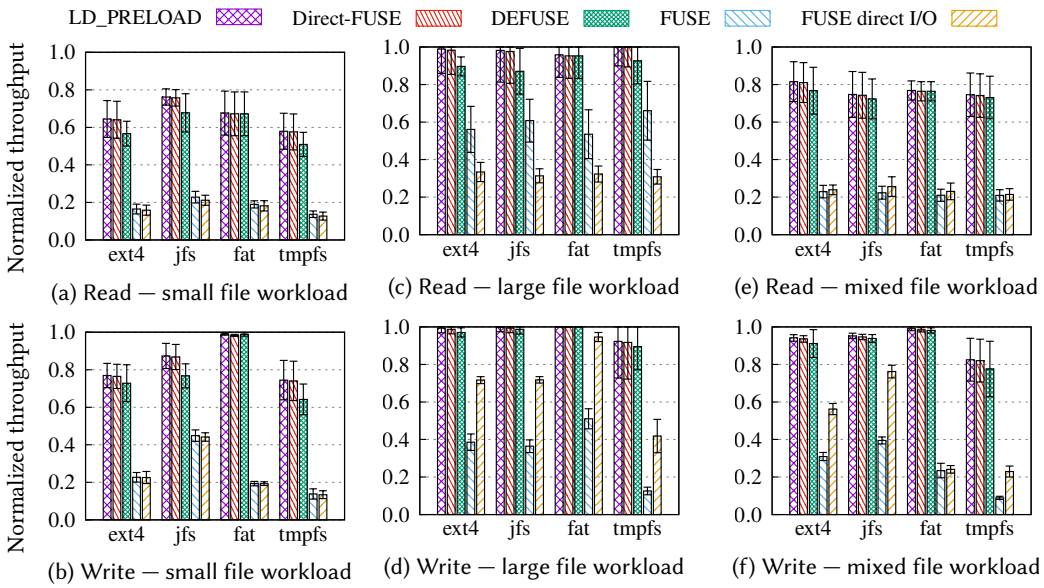

Fig. 7. I/O throughputs of DEFUSE vs an LD_PRELOAD library, Direct-FUSE, and FUSE with and without direct I/O, normalized by the throughput of direct mounted kernel FS (higher is better). (a), (c) and (e) depict read throughput for small, large and mixed file sizes, (b), (d) and (f) depict their respective write counterparts.

also slower than `ext4` with 130 kB/s write and 6 MB/s read throughputs, (d) `tmpfs` I/O operations being completely in memory, achieved the best throughput with 20.4 MB/s for both write and read. However, given the large number of system calls needed to process all the files, achievable throughput is far below the capability of the storage media.

Large files. We evaluated the FSs for large file accesses where the fewest quantity of system calls were needed. A single 128 MB file was created. This benchmark highlights the amortization of wait context switch cost for FUSE's system calls.

The results in Figure 7c and Figure 7d show that DEFUSE reached close to optimal throughputs: between 85% and 93% that of direct mount for reads, and between 92% and 100% for writes. `LD_PRELOAD`'s throughput was almost always above 98% (except for reads on FAT) and, as with small files, Direct-FUSE closely followed. While FUSE performance improved with large files, it remained largely below that of the other interfaces and at best only reached 60% for `tmpfs`. Write performance, on the other hand, almost reached that of direct mount when used with direct I/O for FAT. Since only one file is accessed, the number of system calls serviced by the kernel and FUSE server is small and the total operation time is more dependent on the I/O time to the storage media.

The actual I/O throughput for direct mount is very similar for most FSs, achieving ≈200 MB/s write and ≈900 MB/s read throughput. The exception is `tmpfs` which achieved 2.3 GB/s write and read throughput due to I/O operations being completely in memory.

Mixed sized files. We evaluated the FSs using mixed sized files to simulate a more diverse workload. The benchmark created 2,048 files linearly distributed in sizes from 64 B to 128 kB. Figure 7e and Figure 7f show close results in all cases for all interfaces except FUSE that performs significantly worse. As with previous benchmarks, direct I/O slightly improves FUSE, but is still 15% slower than DEFUSE at best (writes to JFS).

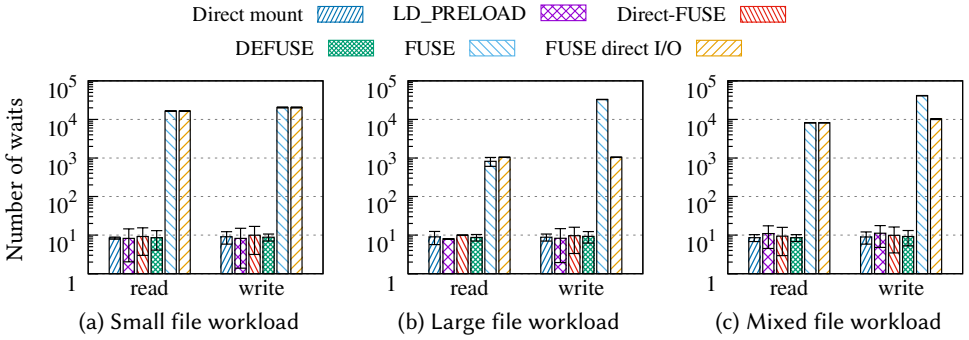

Fig. 8. Number of waits reported by the command `time %w` for the `tmpfs` workloads of Figure 7 (log scale, smaller is better). Using `tmpfs` as backing store removes the I/O overhead inherent to physical media. For (a), thousands of system calls are made but these process-blocking system calls do not always induce a wait.

6.1.3 Wait context switch results. Our evaluation with small, large, and mixed file workloads show that the throughput of DEFUSE was significantly higher than FUSE and was often close to that of `LD_PRELOAD`. We then evaluated the total number of waits needed to perform each of these benchmarks to confirm that the reduced throughput is directly related to context switching and not to any other bottleneck in FUSE. We thus used the Linux `time` command which has the ability to retrieve statistics including the number of context switches performed for both time slice expiration and the purpose of waiting for I/O operations to complete (using `time %w` [52]). Here, we are only concerned about the subset of context switches made by the process for the latter case, i.e., waits. We ran the benchmarks on `tmpfs` to focus on the context switches caused by the FS interface since I/O requests to access a physical device induce extra context switches.

The results shown in Figure 8 (y-axes use a log scale) confirm that the overhead of FUSE is caused in a large part by waits. Direct mount, the `LD_PRELOAD` library, Direct-FUSE and DEFUSE require ≈9 waits for all workloads while FUSE requires orders of magnitude more. For the small file workload (cf. Figure 8a), FUSE requires ≈16,000 waits (i.e., permission `lookup` + `open` + `read`/`write` + `close` for each of the 4,096 files) and using direct I/O does not reduce this number. For the large file workload (cf. Figure 8b), we note that the FUSE kernel driver splits large writes into multiple requests sent to the FUSE server in user space, each causing a wait. Even with a single (large) file, FUSE requires ≈1,000 waits for a read and ≈33,000 for a write. Direct I/O reduces these considerably, from ≈33,000 down to ≈1,000 for writes, which is still 100× higher than DEFUSE. The mixed file workload (cf. Figure 8c) shows an even larger difference in the required waits: ≈9 for DEFUSE but ≈41,000 for FUSE. While there are half as many files created in this workload compared to the small file one (2,048 vs 4,096), files are larger on average thus forcing the FUSE server to split write requests into smaller ones. As with the large file workload, the number of waits is reduced using direct I/O, down to ≈10,000, which is still 1,000× more than DEFUSE.

6.1.4 Application runtime. In addition to synthetic workloads generated by IOzone, we also evaluated DEFUSE's impact on three practical (throughput-intensive) applications using the sources of the Linux kernel v4.15 as data set. First, we decompressed the 160 MB archive comprised of 67,000 files into an `ext4` backed FS using the `tar` command.

We then backed up the decompressed small files from the `ext4` backed FS to a JFS backed FS using `rsync`. We finally compiled the source files using `gcc` on the `ext4` backed FS.

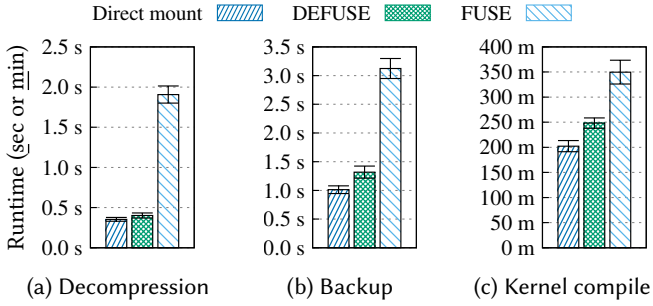

Fig. 9. Runtimes of three throughput-intensive applications using the source files of the Linux kernel v4.15 (lower is better). (a) depicts the time to decompress the 160 MB archive of 67,000 files. (b) depicts the time to back up the decompressed files using `rsync`. (c) depicts the time to compile the kernel using `gcc`.

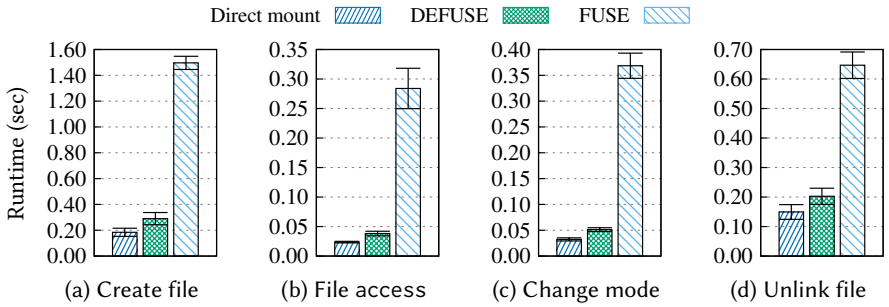

Fig. 10. Runtimes of four benchmarks accessing the metadata of 10,000 files (lower is better). (a) depicts the time to create the files. (b) depicts the time to run the `access` system call on the files. (c) depicts the time to change the files' mode with `chmod`. (d) depicts the time to remove (unlink) the files.

Figure 9a shows the decompression time, Figure 9b the backup time, and Figure 9c the kernel compilation time. For decompression, DEFUSE achieved almost identical runtime to the direct mount while FUSE required as much as 4× the amount of time to finish. Using direct I/O only marginally improved FUSE performance. Similarly, DEFUSE ran at near direct kernel mount speeds when backing up, while FUSE and FUSE with direct I/O were significantly slower, taking 3× longer. For compilation, where DEFUSE and FUSE perform the closest, DEFUSE took 1.23× longer than direct mount and FUSE yet 1.41× longer than DEFUSE, on average. This evaluation showed the practical advantages of DEFUSE over FUSE as a user space FS interface for everyday tasks.

6.1.5 Metadata operations. To complete our single-machine evaluation, we ran benchmarks using various system calls that read, write, and/or modify the metadata of 10,000 files. Unlike the previous benchmarks, these ones are not throughput-intensive but do perform a lot of small I/O operations.

Figure 10a shows the time to create 10,000 files, Figure 10b shows the time to run the `access` system call on all 10,000 files, Figure 10c the time to change the mode of all 10,000 files using the `chmod` system call, and Figure 10d the time to remove (unlink) the 10,000 files. Due to the relatively short amount of time to perform the metadata operation compared to the overhead of the FS access method, FUSE performs significantly worse than direct mount and DEFUSE. Even at its best, for unlink, FUSE requires 4.3× longer than direct mount compared to DEFUSE which takes 1.3× longer

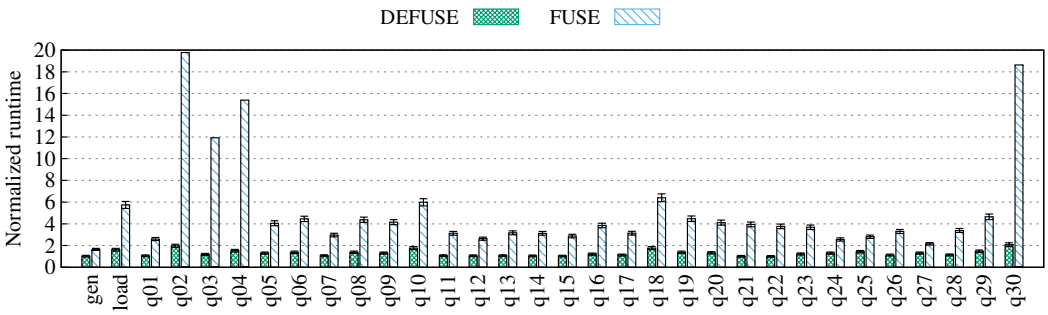

Fig. 11. Runtime of TPCx-BB queries normalized by the runtime of direct mount (lower is better). gen is the time to generate the benchmark data, load the time to load that data into the meta store, qx the time to run query x. DEFUSE is between 1.6× (gen) and 10× (q02) faster than FUSE, and 3.7× on average.

than direct mount; while at its worst FUSE requires 12× longer than direct mount for the file access benchmark where DEFUSE requires 1.6× longer.

6.2 Benefits for Distributed Systems

We also evaluated a diverse set of *distributed* cloud workloads to compare DEFUSE, FUSE, and direct kernel mount. We used Apache Spark [10] v2.4.0 on a cluster of 5 machines with identical hardware setups as described in Section 6.1.1. These distributed workloads show the local benefits of DEFUSE have important *global impact* on distributed applications despite network latency.

TPCx-BB express benchmark. In our first distributed evaluation, we ran the TPCx-BB benchmark suite [91]. The benchmark consists of 30 different SQL queries in the context of retail stores. Using the Spark SQL [11] implementation provided by the Transaction Processing Performance Council, we used a data scaling factor of 100 and used direct mount, DEFUSE, and FUSE as the interfaces of the HDFS backing store. The runtimes shown in Figure 11 are a normalized ratio of HDFS backed by a direct mount `ext4` FS. The entries for `gen` and `load` show the time to generate benchmark data and to load it into the metastore, respectively. Each qx shows the time to run query x. Queries `q22` and `q30` show the best and worst performance respectively for DEFUSE, while queries `q27` and `q02` show the same for FUSE. While the results greatly vary, DEFUSE always outperforms FUSE; DEFUSE is between 1.6× (`gen`) and 10× (`q02`) faster with a 3.7× gain on average for a query.

Word count. To further evaluate DEFUSE against FUSE and a direct mounted FS, we ran a Spark word count workload using 300 GB of Wikipedia data. While Spark data inputs and results may be stored locally or in a remote file system, intermediate results from the map and reduce tasks are stored locally on compute nodes. Using the Purdue University MapReduce benchmarks suite (PUMA) [5] we show that the overall runtime of a Spark application can be affected by the FS interface of this local storage. Figure 12a shows that FUSE significantly increases the completion time of the word count job, even when used only for Spark local storage, with a 2× difference compared to DEFUSE or direct mount. Spark uses memory mapped I/O for reading and writing to local storage so it is not possible to evaluate FUSE with direct I/O enabled.

Machine learning. To evaluate machine learning related workloads, we ran both a linear regression in Spark with a 720 million observation data set (Figure 12b) and k-means with a 1.4 billion observation data set (Figure 12c) using the same Spark configuration used for word count. As with Spark word count, using FUSE seriously impacts the total workload time, requiring 1.75× longer for

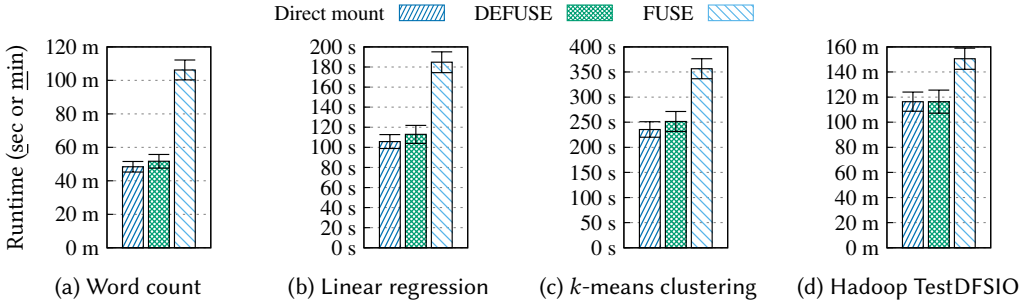

Fig. 12. Runtime of typical cloud workloads (lower is better). (a) depicts the time to run a Spark word count job using 300 GB of Wikipedia data. (b) depicts the time to run a linear regression of 720 million observation data set. (c) depicts the time to run k-means clustering of 200 clusters of a 1.4 billion observation data set. (d) depicts the time to run the TestDFSIO benchmark writing 300 GB of data. DEFUSE is between 1.27× (d) and 1.94× (a) faster than FUSE for cloud workloads, and on par with direct mount for TestDFSIO (d).

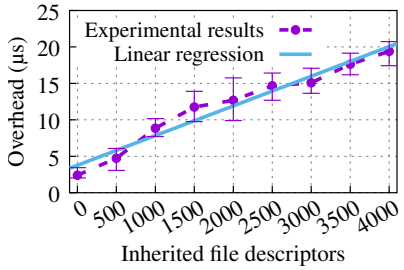

Fig. 13. FD stashing overhead with DEFUSE for up to 4,000 inherited FDs along with the trend line. Overhead is negligible even for 4,000 FDs.

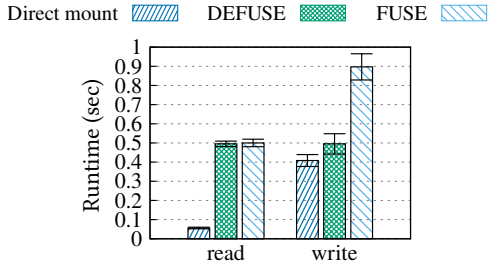

Fig. 14. Runtime of 1.25 M reads and writes on a file mapped with the user space paging of DEFUSE, direct mount FS and FUSE. DEFUSE and FUSE perform equally for reads while writes are 1.8× faster for DEFUSE.

the linear regression workload to complete compared to DEFUSE or direct mount. FUSE's overhead for k-means is lower but is still significant as it takes 1.5× longer than DEFUSE to complete.

HDFS. To conclude our distributed evaluations, we ran the TestDFSIO benchmark to evaluate how HDFS is affected by FS interfaces. The benchmark recorded the runtime of writing a single 300 GB file, plotted in Figure 12d. HDFS uses in-memory data block caching, and performance is affected more by the speed of main memory than by the speed of the FS. However, even in this case, FUSE takes 1.23× longer to complete compared to direct mount or DEFUSE.

6.3 Implementation Microbenchmarks

Last but not least we perform several microbenchmarks to tease apart the savings of DEFUSE.

Overhead of FD stashing. Recall that FD stashing saves and restores the internal FD map used by DEFUSE which adds additional overhead to the `exec` system call. In order to determine the overhead, we created a benchmark where a parent process opens files within DEFUSE and calls `fork` and `exec` to create a child process, then measured and reported in Figure 13 the time for FD stashing to save and restore FDs in the child process. Intuitively, the FD stashing overhead is directly related to the number of inherited FDs as the FD stashing process sequentially saves and

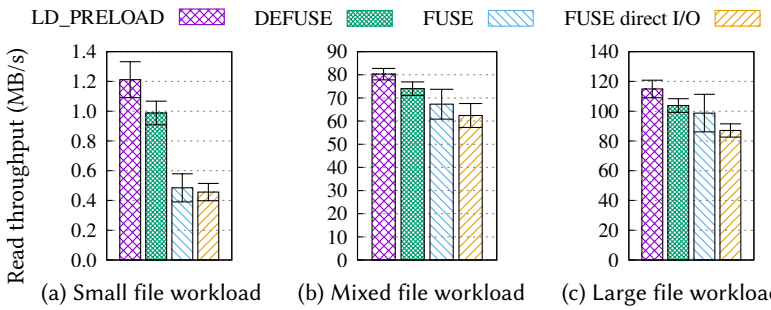

Fig. 15. Read throughput of an LD_PRELOAD library, DEFUSE and FUSE when used with AVFS for small, large and mixed sized file workloads. DEFUSE always performs better than FUSE, with or without direct I/O, and particularly when handling a lot of small files as it reaches much closer to LD_PRELOAD's performance.

restores all FDs. Running a linear regression on the data results in an FD stashing overhead of 4 ns per inherited FD with a base overhead of 3.7 µs to intercept the system calls. Overall, the overhead is virtually nonexistent considering FD stashing is only required once per child process and the typical soft limit for open FDs on Ubuntu Linux is 1,024 (with a total overhead of 8.9 µs).

Overhead of user space paging. To evaluate the performance of DEFUSE user space paging, we evaluated the performance of a file mapped with DEFUSE against a direct mounted FS and FUSE. The comparison does not include FUSE with direct I/O since it is not possible to use it with a memory mapped file — direct I/O bypasses the kernel page cache. We created a benchmark where a 128 MB file is mapped using the `mmap` system call and we then measured and reported in Figure 14 the runtime of 1.25 million random reads and writes over the file's address space. Read operations are significantly slower using either user space access mechanism, DEFUSE or FUSE, than when using direct mount since they both require a transfer of control between kernel and user space for each page fault handled. However, there is no significant difference between the read runtimes of DEFUSE and FUSE. Write operations are 1.8× faster with DEFUSE than with FUSE but 7.5% slower than with direct mount, on average.

AVFS. To further compare LD_PRELOAD, DEFUSE, and FUSE with and without direct I/O enabled, we ran a series of tests on a user space FS implementation that did not have a corresponding kernel driver. Benchmark processes were run for the same small, large, and mixed sized file workloads (cf. Section 6.1.1) using both DEFUSE and the FUSE implementation for AVFS [12] (cf. Section 5.2), a user space only FS that allows direct access to compressed files bypassing the need for a decompression tool. A `gzip` compressed `tar` file was created containing the files of each workload. Figure 15 shows DEFUSE achieved higher read throughput for small files, almost double that of FUSE. Both large and mixed file workloads show less gains, but still clear improvements over FUSE.

7 RELATED WORK

We discuss work related to user space FS interfaces and performance improvements to FUSE.

7.1 Related to User Space Libraries

MPI-IO [55] is developed as an extension to the Message Passing Interface (MPI) with the purpose of improving the performance of collective I/O within a parallel computing system. ROMIO [85] is a high performance user space implementation of MPI-IO. Its use of an abstract interface [84] allows multiple backend FSs to be connected to the library allowing parallel applications written to

the MPI standard to access the underlying FSs without specific knowledge of FS API. Operating completely in user space, this solution provides the benefits of FS access speed, while the abstract interface allows for FS-agnostic access. However, it is intended for use with parallel computing systems and applications must be coded specifically to the MPI-IO standard. Applications using POSIX system calls would not be able to access the library.

Like MPI-IO, `libsysio` [89] is another library initially developed for managing access to FSs for high performance computing. However, as with other directly linked user space libraries, it is affected by the FD heritage dilemma nor does it provide user space paging.

ADAPT [94] provides an auxiliary storage data path to improve performance and flexibility while being compatible with existing storage models. Data is associated with one or more tags describing the relationships between data and then uses a single-level store provided via a key-value system for data access. It is implemented primarily as user space library allowing applications aware of its API access to its capabilities, however also offers a FUSE implementation for legacy applications.

7.2 Related to FUSE

Numerous solutions have improved FUSE but only few tackle the basic interface overhead largely due to wait context switch latency, without providing POSIX compliance.

Ishiguro et al. [40] propose a method to reduce memory copies and wait context switches when FUSE is used with a remote FS. Their *direct device access* prevents unneeded calls from the FUSE kernel driver to user space when the backing FS contains a kernel driver. However, this solution is not intended for FSs whose I/O request handling logic resides in user space.

ExtFUSE [16] proposes to reduce the number of FUSE requests between kernel space and user space using extended Berkeley packet filters (eBPF) [50]. The eBPF interface allows for user space programs to run in kernel space with significant restrictions. While ExtFUSE can reduce the number of wait context switches, it is isolated to a subset of I/O operations.

Direct-FUSE [100] provides an abstract interface for user space FS libraries within `libsysio` [89]. As mentioned in Section 6.1.1, it does not address the FD heritage dilemma, nor does provide an interface for user space paging, and therefore does not provide all the guarantees required by POSIX.

WrapFS [97] provides an efficient *wrapper* of a file system mount onto another mounted location. However it does not provide the bypassed FD lookup behavior needed to reduce the time to service an open system call.

Stacked FSs [70] promise ease in deployment of incremental changes to existing FSs. File System Translator (FiST) [98] is a tool aiming to ease creation of stacked FSs by generating the necessary kernel drivers for FS drivers written in a high-level language. However, both stacked FSs and FiST are limited to kernel space FS drivers. Narayan et. al. [60] extend FiST for user space implementations but in a way dependent on FUSE, thus suffering from wait context switch overhead.

Ganesha network file system (NFS) [24] is a user space NFS client supporting the same protocols as kernel space implementation as in other Unix-like OSs. Yet running in user space allows Ganesha NFS to redirect I/O requests directly to user space FS libraries using an FS abstraction layer [62] similar to FUSE's. While increasing flexibility for NFS exports, it does not improve performance of interaction between kernel and user spaces; the socket interface used by the RPC protocol of NFS has similar performance penalties as the FUSE interface used to communicate between kernel and user space.

Steere et al. [78] propose a caching mechanism running in user space as an optimization to the Coda distributed FS [72]. While the cache improves performance for I/O operations, the system still requires a transfer of control from kernel to user space for *each* directory tree level during lookup. Even with the existence of a user space cache, workloads using input data sets requiring

access to potentially millions of small files [17, 21] would take a large number of control transfers to fill the cache, yielding performance comparable to FUSE.

Patlasov [65] sketches a set of performance optimizations for FUSE aiming at parellelization thanks to, e.g., multi-threading, direct I/O, and caching. However, wait context switches are not reduced, the optimizations are only validated using specialized parallel workloads on specialized iSCSI SAN storage rather than on common FUSE, and no codebase is public.

Re-FUSE [79] (which is not ReFUSE [44]) is an extension to FUSE to make the user space FUSE server fault-tolerant. A Re-FUSE server crash is automatically detected and a restart initiated while applications continue to run without knowledge of the failure. FUSE performance is not improved.

7.3 Related to LD_PRELOAD

OrangeFS Direct Interface [64] uses an LD_PRELOAD implementation, but it experiences the FD heritage dilemma. Furthermore, its implementation is tightly coupled with PVFS [71] thus it is not practical as a general purpose FS.

Goanna [77] is a framework for rapid FS implementation in user space based on the ptrace kernel interface. ptrace allows for a process to monitor for, and intercept system calls of, other processes. Debuggers such as gdb [32] use this interface to set breakpoints in executing processes. When a system call from an FS is intercepted, Goanna redirects the operation to a user space FS library as appropriate. This has a similar effect to an LD_PRELOAD library without losing internal mappings when a parent creates a child process with open FDs. However, using ptrace requires similar wait context switches between kernel and user space as processes FUSE. Goanna mitigates these by using a modified ptrace implementation, however still leaving a large number of wait context switches. Also, for Goanna to monitor all processes, it must run with superuser privileges, which is a security risk even if it runs in user space.

SplitFS [43] is an FS for persistent memory that is uses an LD_PRELOAD implementation to intercept FS related system calls. It contains an implementation similar to FD stashing however does not improve the performance of open as DEFUSE does though bypassed FD lookup. For applications that manipulate a large number of small files, the overhead for open can dwarf the overhead for doing the I/O operations themselves. Such is the case for high performance computing workloads, some of which create millions of files below 64 kb. In addition, SplitFS is specialized for use with persistent memory and is not a general purpose user space FS interface.

Bypass and Multiple Bypass [83] use LD_PRELOAD to ease the writing of distributed applications. Calls intercepted by an LD_PRELOAD library may be executed locally or redirected to a remote machine, e.g., to read data on a remote FS, depending on the policies defined in Bypass. This scheme enables applications to split their execution on multiple machines with low programming effort.

7.4 Related to User Space Paging

FluidMem [20] is a library designed to provide memory resource disaggregation by creating a system for memory as a service. It uses the userfaultfd call to manage page faults, however its infrastructure acts as a service within a hypervisor providing access to remote memory for a client virtual machine by accessing memory pages on a remote system.

UMap [66] is a library to provide application-driven optimizations for page management. It uses userfaultfd to control access to a page map providing applications fine-grained control of page management (e.g., pre-fetching and controlled flushing of pages), however it is not an application-agnostic interface and requires client applications to make explicit calls to set the policies for page management.

8 CONCLUSIONS

FSs implemented in user space have the advantage of permission isolation, access to user space libraries, and ease of prototyping using a diverse set of programming languages. While deployable interfaces for user space FSs exist, they suffer from significant performance penalties (e.g., FUSE) and/or inconsistent behaviors FDs (e.g., `LD_PRELOAD`-loaded library). Research attempting to improve access performance for user space FSs is often workload-specific (e.g., MPI-IO) or does not address inconsistency issues (e.g., Direct-FUSE).

DEFUSE is a generic interface for user space FSs that enables FS access using existing POSIX system calls with higher access speed than FUSE, up to 2× faster for persistent storage media and as high as 10× faster for memory-based operations in our evaluation. At the same time, DEFUSE improves on existing user space FS interfaces by maintaining consistency of FDs passed between parent and child processes. In future work, we plan to further improve performance of DEFUSE and provide a wrapper interface to streamline porting existing FS implementations from FUSE into DEFUSE such that existing FUSE implementations can be used with DEFUSE with no modification.

REFERENCES

[1] Personal conversation with David Bonnie, storage tech lead at Los Alamos National Laboratory and co-designer of OrangeFS/PVFS2, in reference to work on MarFS 11/15/2016.

[2] Access DBFS using local file APIs. https://docs.databricks.com/user-guide/dbfs-databricks-file-system.html#access-dbfs-using-local-file-apis.

[3] Access DBFS with the Databricks CLI. https://docs.databricks.com/user-guide/dbfs-databricks-file-system.html#access-dbfs-with-the-databricks-cli.

[4] AccessFS: Permission Filesystem for Linux. http://www.olafdietsche.de/2002/11/07/accessfs-permission-filesystem-linux/.

[5] Ahmad, F., Lee, S., Thottethodi, M., and Vijaykumar, T. PUMA: Purdue University Benchmark Suite, 2012.

[6] Alluxio-FUSE. https://github.com/Alluxio/alluxio/tree/master/integration/fuse.

[7] Amazon S3. https://aws.amazon.com/s3/.

[8] Amazon S3 FUSE. https://github.com/s3fs-fuse/s3fs-fuse.

[9] Apache Hadoop 2.4.1 - File System Shell Guide. https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-common/FileSystemShell.html#Overview.

[10] Apache Spark. http://spark.apache.org/.

[11] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)* (2015), pp. 1383–1394.

[12] AVFS - A Virtual File System. http://avf.sourceforge.net/.

[13] Amazon Web Services SDK for C++. https://aws.amazon.com/sdk-for-cpp/.

[14] Behrens, D., Serafini, M., Junqueira, F. P., Arnautov, S., and Fetzer, C. Scalable Error Isolation for Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)* (2015), pp. 605–620.

[15] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. PLFS: A Checkpoint Filesystem for Parallel Applications. In *High Performance Computing, Networking, Storage and Analysis (SC '09)* (2009), pp. 1–12.

[16] Bijlani, A., and Ramachandran, U. Extension Framework for File Systems in User Space. In *2019 USENIX Annual Technical Conference (ATC '19)* (2019), pp. 121–134.

[17] Bonnie, D. J., and Torres, A. G. Small File Aggregation with PLFS. Tech. rep., Los Alamos National Laboratory (LANL), 2013.

[18] Borthakur, D., et al. HDFS architecture guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2008.

[19] Burihabwa, D., Felber, P., Mercier, H., and Schiavoni, V. SGX-FS: Hardening a File System in User-Space with Intel SGX. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom '18)* (2018), pp. 67–72.

[20] Caldwell, B., Goodarzy, S., Ha, S., Han, R., Keller, E., Rozner, E., and Im, Y. FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud. In *Proceedings of the 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS'20)* (2020), pp. 665–677.

[21] Carns, P., Lang, S., Ross, R., Vilayannur, M., Kunkel, J., and Ludwig, T. Small-File Access in Parallel File Systems. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS '09)* (2009), pp. 1–11.

[22] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., Snir, M., Traversat, B., and Wong, P. Overview of the MPI-IO Parallel I/O Interface. In *Workshop on Input/Output in Parallel and Distributed Systems (IPPS '95)* (1995), pp. 1–15.
[23] Databricks File System. https://docs.databricks.com/user-guide/dbfs-databricks-file-system.html/.
[24] Deniel, P., Leibovici, T., and Lafoucrière, J.-C. GANESHA, A Multi-Usage with Large Cache NFSv4 Server. In *Linux Symposium* (2007), p. 113.
[25] Emacs Hooks. https://www.gnu.org/software/emacs/manual/html_node/emacs/Hooks.html.
[26] Essertel, G., Tahboub, R., Decker, J., Brown, K., Olukotun, K., and Rompf, T. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)* (2018), pp. 799–815.
[27] Ext4 (and Ext2/Ext3) Wiki. https://ext4.wiki.kernel.org/.
[28] FAT Filesystem Library in R6RS Scheme. https://gitlab.com/weinholt/fs-fatfs.
[29] FUSE Example fusexmp. https://github.com/fuse4x/fuse/blob/master/example/fusexmp.c.
[30] FUSE for Google Cloud Storage. https://github.com/GoogleCloudPlatform/gcsfuse/.
[31] FUSE High Level Interface. https://github.com/libfuse/libfuse/blob/master/include/fuse.h.
[32] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.
[33] GlusterFS - A Scale-Out Network-Attached Storage File System. https://www.gluster.org/.
[34] Google Cloud Storage. https://cloud.google.com/storage/.
[35] gsutil tool. https://cloud.google.com/storage/docs/gsutil.
[36] Hupfeld, F., Cortes, T., Kolbeck, B., Stender, J., Focht, E., Hess, M., Malo, J., Marti, J., and Cesario, E. XtreemFS: A Case for Object-Based Storage in Grid Data Management. In *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB* (2007).
[37] IBM Spectrum Scale - Formerly General Parallel File System (GPFS). https://www.ibm.com/us-en/marketplace/scale-out-file-and-object-storage.
[38] Inman, J. T., Vining, W. F., Ransom, G. W., and Grider, G. A. MarFS, a Near-POSIX Interface to Cloud Objects. *The USENIX Magazine* (Spring 2017).
[39] IOzone Filesystem Benchmark. http://iozone.org/.
[40] Ishiguro, S., Murakami, J., Oyama, Y., and Tatebe, O. Optimizing Local File Accesses for FUSE-based Distributed Storage. In *High Performance Computing, Networking, Storage and Analysis (SC '12)* (2012), pp. 760–765.
[41] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., et al. BetrFS: A Right-Optimized Write-Optimized File System. In *13rd USENIX Conference on File and Storage Technologies (FAST '15)* (2015), pp. 301–315.
[42] Journaled File System Technology for Linux. http://jfs.sourceforge.net/.
[43] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A., and Chidambaram, V. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)* (2019), pp. 494–508.
[44] Kantee, A., and Crooks, A. ReFUSE: Userspace FUSE Reimplementation using PUFFS. In *6th European BSD Conference (EuroBSDCon '07)* (2007).
[45] Leslie, B., Chubb, P., Fitzroy-Dale, N., Götz, S., Gray, C., Macpherson, L., Potts, D., Shen, Y.-T., Elphinstone, K., and Heiser, G. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology 20*, 5 (Sep 2005), 654–664.
[46] Li, H. *Alluxio: A Virtual Distributed File System.* PhD thesis, UC Berkeley, 2018.
[47] C API libhdfs. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/LibHdfs.html.
[48] libfuse - Filesystem in Userspace. https://github.com/libfuse/libfuse.
[49] libfuse - SSHFS implementation. https://github.com/libfuse/sshfs.
[50] Linux Manual - bpf - Perform a Command on an Extended BPF Map or Program. http://man7.org/linux/man-pages/man2/bpf.2.html.
[51] Linux Manual - Overview, Conventions, and Miscellaneous: libc. http://man7.org/linux/man-pages/man7/libc.7.html.
[52] Linux User Manual - Time Command, Option %w for Waits. https://linux.die.net/man/1/time.
[53] Linux Virtual File System. http://www.tldp.org/LDP/tlk/fs/filesystem.html.
[54] Lustre Parallel File System. http://lustre.org/.
[55] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.
[56] Microsoft Corporation. Microsoft Extensible Firmware Initiative FAT32 File System Specification. Tech. rep., Microsoft Corporation, 2000.
[57] Moose File System (MooseFS). https://moosefs.com/index.html.
[58] Mountable HDFS. https://wiki.apache.org/hadoop/MountableHDFS.
[59] Muniswamy-Reddy, K.-K., Wright, C. P., Himmer, A., and Zadok, E. A Versatile and User-Oriented Versioning File

System. In *3rd USENIX Conference on File and Storage Technologies (FAST '04)* (2004), vol. 4, pp. 115–128.
[60] Narayan, S., Mehta, R. K., and Chandy, J. A. User Space Storage System Stack Modules with File Level Control. In *Proceedings of the 12th Annual Linux Symposium in Ottawa* (2010), pp. 189–196.
[61] Native HDFS FUSE. https://github.com/remis-thoughts/native-hdfs-fuse.
[62] NFS Ganesha - File System Abstraction Layer (FSAL). https://github.com/nfs-ganesha/nfs-ganesha/wiki/Fsalsupport.
[63] ObjectiveFS. https://objectivefs.com/.
[64] OrangeFS Direct Interface. http://docs.orangefs.com/v_2_9/Direct_Interface.htm.
[65] Patlasov, M. Optimizing FUSE for Cloud Storage. In *Linux Vault* (2015).
[66] Peng, I., McFadden, M., Green, E., Iwabuchi, K., Wu, K., Li, D., Pearce, R., and Gokhale, M. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC '19)* (2019), IEEE, pp. 71–78.
[67] Pillai, M., Gowdappa, R., and Henk, C. Experiences with Fuse in the Real World. In *2019 Linux Storage and Filesystems Conference (VAULT '19)* (Feb. 2019).
[68] Rajachandrasekar, R., Moody, A., Mohror, K., and Panda, D. K. A 1 PB/s File System to Checkpoint Three Million MPI Tasks. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '13)* (2013), pp. 143–154.
[69] React Hooks. https://reactjs.org/docs/hooks-intro.html.
[70] Rosenthal, D. S. Evolving the Vnode interface. In *USENIX Summer* (1990), vol. 99, pp. 107–118.
[71] Ross, R. B., Thakur, R., et al. PVFS: A Parallel File System for Linux Clusters. In *The 4th Annual Linux Showcase and Conference* (2000), pp. 391–430.
[72] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers 39*, 4 (1990), 447–459.
[73] Sharwood, S. Linux literally loses its Lustre – HPC filesystem ditched in new kernel, 2018.
[74] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *26th IEEE Symposium on Massive Storage Systems and Technologies (MSST '10)* (2010), pp. 1–10.
[75] Solucorp VirtualFS. http://www.solucorp.qc.ca/virtualfs/.
[76] Spark PySpark Daemon. https://github.com/apache/spark/blob/5264164a67df98b73facae207eda12ee133be7d/python/pyspark/daemon.py.
[77] Spillane, R. P., Wright, C. P., Sivathanu, G., and Zadok, E. Rapid File System Development using ptrace. In *Workshop on Experimental Computer Science, Part of ACM FCRC* (2007), p. 22.
[78] Steere, D. C., Kistler, J. J., and Satyanarayanan, M. Efficient User-Level File Cache Management on the Sun vnode Interface. In *USENIX Summer* (1990), vol. 99, pp. 325–332.
[79] Sundararaman, S., Visampalli, L., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Refuse to Crash with Re-FUSE. In *6th Conference on Computer Systems (EuroSys '11)* (2011), pp. 77–90.
[80] System-call wrappers for glibc. https://lwn.net/Articles/799331/.
[81] Tahoe-LAFS - Tahoe Least-Authority File Store. https://tahoe-lafs.org/trac/tahoe-lafs/.
[82] Tarasov, V., Gupta, A., Sourav, K., Trehan, S., and Zadok, E. Terra Incognita: On the Practicality of User-Space File Systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '15)* (2015).
[83] Thain, D., and Livny, M. Multiple Bypass: Interposition Agents for Distributed Computing. *Cluster Computing 4*, 1 (2001), 39–47.
[84] Thakur, R., Gropp, W., and Lusk, E. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *6th Symposium on the Frontiers of Massively Parallel Computing (Frontiers '96)* (1996), pp. 180–187.
[85] Thakur, R., Lusk, E., and Gropp, W. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Tech. rep., Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.
[86] The Linux Kernel - d_splice_alias. https://www.kernel.org/doc/htmldocs/filesystems/API-d-splice-alias.html.
[87] The Linux Kernel - userfaultfd. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html.
[88] The Plastic File System. http://plasticfs.sourceforge.net/.
[89] SYSIO Library. https://libsysio.sourceforge.io/.
[90] tmpfs Documentation. https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt.
[91] TPCx-BB Specification. https://www.tpc.org/.
[92] User-space page fault handling. https://lwn.net/Articles/550555/.
[93] Vangoor, B. K. R., Agarwal, P., Mathew, M., Ramachandran, A., Sivaraman, S., Tarasov, V., and Zadok, E. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Transactions on Storage 15*, 2 (May 2019).
[94] Wang, W., Meyers, C., Roy, R., Diesburg, S., and Wang, A.-I. A. ADAPT: An auxiliary storage data path toolkit.

Journal of Systems Architecture 113 (2021), 101902.

[95] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A Scalable, High-Performance Distributed File System. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)* (2006), pp. 307–320.

[96] Wright, S. A., Hammond, S. D., Pennycook, S. J., Miller, I., Herdman, J. A., and Jarvis, S. A. LDPLFS: Improving I/O Performance without Application Modification. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12)* (2012), pp. 1352–1359.

[97] Zadok, E., and Bădulescu, I. A Stackable File System Interface for Linux. In *LinuxExpo Conference Proceedings* (May 1999), pp. 141–151.

[98] Zadok, E., and Nieh, J. FiST: A Language for Stackable Filesystems.

[99] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12)* (2012), pp. 15–28.

[100] Zhu, Y., Wang, T., Mohror, K., Moody, A., Sato, K., Khan, M., and Yu, W. Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support. In *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '18)* (2018), p. 6.