Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support

Yue Zhu
Florida State University
Tallahassee, Florida
yzhu@cs.fsu.edu

Teng Wang
Florida State University
Tallahassee, Florida
twang@cs.fsu.edu

Kathryn Mohror
Lawrence Livermore National Laboratory
Livermore, California
kathryn@llnl.gov

Adam Moody
Lawrence Livermore National Laboratory
Livermore, California
moody20@llnl.gov

Kento Sato
Lawrence Livermore National Laboratory
Livermore, California
sato5@llnl.gov

Muhib Khan
Florida State University
Tallahassee, Florida
khan@cs.fsu.edu

Weikuan Yu
Florida State University
Tallahassee, Florida
yuw@cs.fsu.edu

ABSTRACT

Developing a file system is a challenging task, especially a kernel-level file system. User-level file systems alleviate the burden and development complexity associated with kernel-level implementations. The Filesystem in Userspace (FUSE) is a widely used tool that allows non-privileged users to develop file systems in user space. When a FUSE file system is mounted, it runs as a user-level process. Application programs and FUSE file system processes are bridged through FUSE kernel module. However, as the FUSE kernel module transfers requests between an application program and a file system process, the overheads in a FUSE file system call from crossing the user-kernel boundary is non-trivial. The overheads contain user-kernel mode switches, context switches, and additional memory copies. In this paper, we describe our Direct-FUSE framework that supports multiple FUSE file systems as well as other, custom user-level file systems in user space without the need to cross the user/kernel boundary into the FUSE kernel module. All layers of Direct-FUSE are in user space, and applications can directly use pre-defined unified file system calls to interact with different user-defined file systems. Our performance results show that Direct-FUSE can outperform some native FUSE file systems by 11.9% on average and does not add significant overhead over backend file systems.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Runtime environments**;

KEYWORDS

FUSE, File System, User-level File System

ACM Reference Format:
Yue Zhu, Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, Muhib Khan, and Weikuan Yu. 2018. Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support. In *ROSS'18: 8th International Workshop on Runtime and Operating Systems for Supercomputers, June 12, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3217189.3217195

1 INTRODUCTION

An efficient file system is important for high-performance computing (HPC) systems ([1, 3, 4, 7]) in supporting large-scale scientific applications. In general, kernel-level file systems are generalized for addressing the broader Linux market, while user-level file systems are designed with special-purpose features for particular I/O workloads. The lack of specialization in kernel-level file systems is due to the complexities of development at the kernel level and the risk of interacting with the kernel including kernel code security, maintainability (people change), and coding style.

The Filesystem in Userspace (FUSE) [25] is a software framework for Unix-like file systems, which allows non-privileged users to create their file systems without kernel-based file system implementations. A FUSE file system is usually achieved as a standalone application linked with the *libfuse* library. The *libfuse* library provides a reference implementation for communicating with the FUSE kernel module and mounting/unmounting file systems [25]. The file system functions are implemented in the *libfuse* library as callbacks. A FUSE file system runs as a user process, and its file system calls to the mount point are forwarded to the process via the FUSE kernel module. SSHFS [24], GmailFS [12], FusionFS [32], and clients of GlusterFS [9] are well-known FUSE file systems. There are also other user-level file systems that do not use the FUSE framework, such as CRUISE [18], BurstFS [29], and DeltaFS [33]. They intercept the application I/O via a set of wrapper functions around the POSIX I/O calls or by defining their own file system API.

Most user-level file systems are designed to support particular I/O workloads. When needed, multiple different file systems can be used for different kinds of data in a single job. For example, there could be an in-memory file system for checkpoints (e.g. CRUISE), a file system for efficiently managing metadata (e.g. TableFS [21], MetaKV [30]), and a file system for shared files across burst buffers (e.g. BurstFS). However, interacting with multiple user-level file systems is a challenging task. Since FUSE file systems run as processes in user space, the communication round trip between an application program and file system processes could raise non-trivial overheads affecting application performance. To mitigate the overheads, some FUSE file systems circumvent the involvement of the FUSE kernel [9, 14, 21, 27, 32]. However, not all FUSE file systems enable the FUSE kernel circumvention, and it is arduous for users to bypass the FUSE kernel for the file systems and also collaborate with others. In addition, root permission is required to mount FUSE file systems, or system administrators have to give non-privileged users the ability to mount FUSE file systems. On HPC systems, users do not typically have superuser privileges and cannot easily mount the desired FUSE file systems without the help from system administrators.

In this paper, we describe Direct-FUSE, a framework to support user-level file systems without crossing the kernel boundary. Direct-FUSE is built on top of *libsysio* [5], which is developed by Sandia Scalable I/O team and provides a POSIX-like interface which redirects I/O function calls to targeted remote file systems. Direct-FUSE is designed to support various user-level file systems in one job with a unified POSIX-like interface, numerous backends, and fewer overheads. In this work, we make the following contributions:

(1) We evaluate the overheads introduced by the round trips in FUSE file system calls and give a detailed cost breakdown analysis.
(2) We describe the design of Direct-FUSE, which facilitates the capability of using multiple user-level file systems as backends for different I/O workloads.
(3) We evaluate Direct-FUSE performance with other FUSE local or distributed file systems. We also carry out a cost analysis for Direct-FUSE and compare it with native file systems and FUSE file systems.

The paper is organized as follows. In Section 2, we present an overview of FUSE file systems and libsysio, conduct a cost analysis for FUSE file systems, and introduce current methods for reducing FUSE overheads. In Section 3, we describe the design of Direct-FUSE and the implementation details in incorporating a new user-level file system. After presenting our experimental results and cost analysis in Section 4, we summarize related work in Section 5. We end the paper by drawing our conclusion in Section 6.

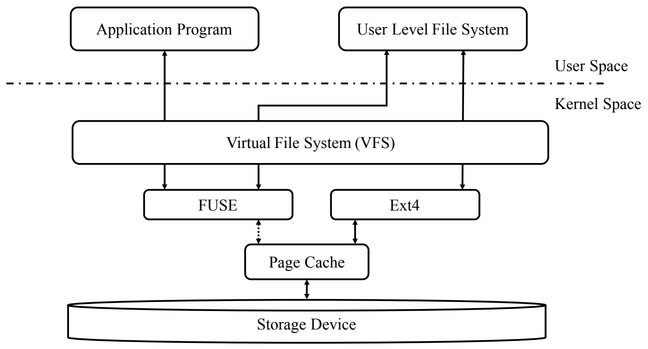

Figure 1: Path of a FUSE file system call.

2 BACKGROUND & MOTIVATIONS

In this section, we present the background of FUSE in Section 2.1.1, the introduction of libsysio [5] in Section 2.1.2, two FUSE overhead analyses in Section 2.2 and Section 2.3, and some current methods of reducing the overheads in Section 2.4.

2.1 Background

2.1.1 FUSE - Filesystem in Userspace. Normally, a FUSE file system run as a user-level process referred to as *userfs* for convenience. *userfs* communicates with application program through FUSE kernel module which is located in the kernel space. Once a *userfs* is mounted by users, the FUSE kernel module registers the file system with the Virtual File System (VFS) and also a block device named /dev/fuse. Any file system calls issued to the mount point are forwarded to the *userfs* through VFS, FUSE module, and /dev/fuse.

Fig. 1 shows the data flow of a file system call in FUSE. We use read() as an example for illustration. After a read call is issued to the mount point from an application program, it is passed to the FUSE kernel module via VFS. If the requested data is in the page cache, the data will be returned immediately. If not, the read call will be forwarded to *userfs* through /dev/fuse, then a user-defined read function will be invoked. Moreover, if the user-defined file system overlays any native file systems, such as Ext4, the read call is continually forwarded to the native file system from *userfs*. The native file system will read data from the storage device and send them to *userfs*. The *userfs* then sends requests to the application program via /dev/fuse block device, FUSE kernel, and VFS [28].

2.1.2 libsysio. Our *Direct-FUSE* is designed to provide multiple backend services for one application without crossing kernel boundary. We build our work on top of *libsysio*.

The critical metadata structures in libsysio are *mount*, *pnode*, *pnode_base*, and *inode. mount* contains information for the mount point, including the pnode of the mount point. *pnode* is an entry in a directory, which has the pointer to *pnode_base*, the pointer to the parent, the pointer to the mount information, etc. *pnode_base* contains the name for the entry in the directory, the inode pointer, etc. An *inode* record is maintained for each file object in the system, which has the pointer to file system operations. Thus, pnode, pnode_base, and inode are chained together; mount is chained with

Table 1: FUSE-tmpfs vs. tmpfs

	# Context Switches		Write Bandwidth	
Block Size (KB)	FUSE-tmpfs	tmpfs	FUSE-tmpfs (MB/s)	tmpfs (GB/s)
4	1012	7	163	1.3
16	1012	7	372	1.6
64	1012	7	519	1.7
128	1012	7	549	2.0
256	2012	7	469	2.4

pnode if the pnode is referred to the pnode of mount point. Therefore, libsysio can walk an absolute path, look up each component related to the path and return the pnode information.

2.2 The Overheads of FUSE File Systems

Although FUSE can alleviate the headache of developing a kernel-based file system, it introduces a lot of overheads for any type of file system calls [19] because of the additional round trips between application programs and *userfs*.

To gain more understanding about the overheads, we carry out a cost analysis for FUSE file system calls. These costs include user-kernel mode switches, context switches, and memory copies. For a typical native file system, such as Ext4, there are only 2 user-kernel mode switches (to and from the kernel), no context switch, and 1 memory copy (kernel page cache ↔ application). However, a file system call in FUSE involves 4 user-kernel mode switches (application ↔ FUSE kernel, FUSE kernel ↔ *userfs*), 2 context switches (scheduler switching between two processes), and possibly more than 2 memory copies (application → page cache → *userfs*, and *userfs* → underlying file system if the FUSE file system is built on top of others). These overheads are the cause of the performance difference between FUSE file systems and native file systems. Especially, the two unavoidable context switches can significantly affect the performance of FUSE file systems. Because when switching between an application process and *userfs*, the OS kernel scheduler is needed, and processor registers, TLB, and cache have to be saved and restored [13].

Table 1 shows an example comparison on both the write performance and the number of context switches of a FUSE file system (*FUSE-tmpfs*) and a native file system (*tmpfs* [22]). *FUSE-tmpfs* is a FUSE file system overlying *tmpfs*, which is a file system storing data on volatile memory. We use *dd* micro-benchmark and *perf* [8] system performance profiling tool for these tests. In specific, we use *dd* to write 1000 data segments to the file system and vary the transfer sizes from 4 KB to 1 MB, respectively. We use *perf* to trace the context switches events happening on the test program side. In addition, because the default data size of FUSE request is 4 KB, when the data size to write exceeds 4 KB, additional FUSE requests have to be sent to *userfs*. To give a better performance of *FUSE-tmpfs*, we enable *max_write* with 128 KB write size per request to reduce the number of FUSE requests exchanged between the test process and *userfs*. As *max_write* sets the maximum write size of one FUSE request to 128 KB, for the size of FUSE write requests lower than 128 KB, these write requests are sent independently; for the write requests' size greater than 128 KB, they have to be split to multiple data chunks. This operation leads to better performance

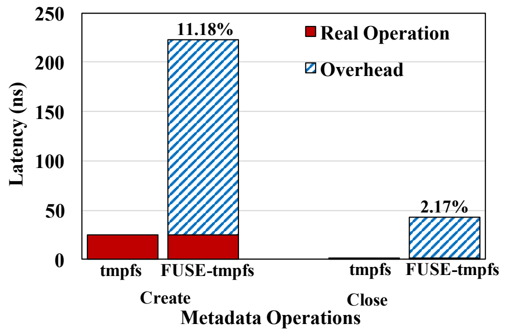

Figure 2: Time expense in metadata operations.

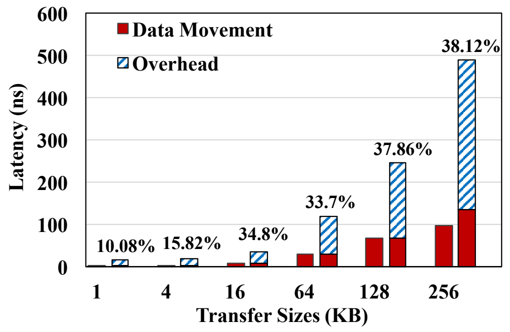

Figure 3: Time expense in data operations (write).

due to less number of mode switches and context switches for large writes. As we only profile the number of context switches on the test program side, other context switches on the FUSE file system process side are not reflected in Table 1.

Therefore, as shown in Table 1, when the transfer size is less than 128 KB, there are 1012 context switches in *FUSE-tmpfs*, including 1000 context switches for 1000 writes, 5 context switches for 5 file system calls (i.e. open(), close(), release() and two flush()), and 7 context switches for kernel scheduling for writing to *tmpfs*. When the transfer size is greater than 128 KB, additional context switches are added, which is shown on the last row of Table 1. Since a *tmpfs* file system call does not involve any round trips, there are no additional context switches besides the 7 constant context switches for kernel scheduling. As the result shows, the write bandwidth of *FUSE-tmpfs* is greatly lower than that of *tmpfs*, caused by the additional context switches and memory copies during the write calls.

2.3 Dissection of FUSE File System Calls

Furthermore, we also investigate a breakdown of time in FUSE metadata and data operations such as *create()*, *close()*, and *write()*. Similarly, we measure the time spent on user-defined operations and file system calls on *FUSE-tmpfs*, and compare the results with file system calls on *tmpfs*. The *Data Movement* and the *Overhead* in Fig. 2 and Fig. 3 indicate the time spent on user-defined operations and the time besides user-defined operations, respectively. As shown in these two figures, the time spent on user-defined operations counts for only a small portion of the total time for a file system call of *FUSE-tmpfs*. For example, in Fig. 2, the time of user-defined

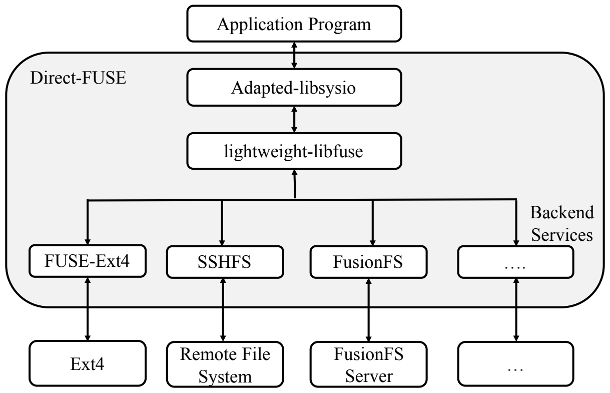

Figure 4: Direct-FUSE file system architecture.

create is only 11.18% of a complete *FUSE-tmpfs* create(), and the user-defined close time only takes 2.17% of the total time for *FUSE-tmpfs* close(). In Fig. 3, we can see that for any transfer sizes, the time of user-defined write() is a small portion of the total execution time. Moreover, when the transfer size is greater than 128 KB, as more FUSE requests are sent through /dev/fuse, the actual write time (*Data Movement*) of *FUSE-tmpfs* is higher than the write time of *tmpfs*.

2.4 Reduce The Overhead from FUSE Kernel

To avoid the overheads from crossing FUSE kernel boundary, some FUSE file systems are allowed to be leveraged as independent user-level libraries, such as TableFS [21] and FusionFS [32]. They have to expose file system operations defined in *fuse_operations* structure of FUSE project to a user-level library, referred to as FUSE user-level library. The fuse_operations structure is exposed to developers to customize file system operations for FUSE high-level API. The difference between the FUSE high-level and low-level API is discussed in Section 3.3. However, as different FUSE file systems may be needed to handle different tasks (e.g., checkpoint, metadata management) in a single job, this approach may not directly support using multiple FUSE user-level libraries in a single job, since there is no distinct boundary to differentiate files for specific libraries while using. More specifically, since multiple FUSE user-level libraries are integrated individually for different purposes, two different FUSE libraries may return same file descriptors for two unique file paths. In this way, users have to pay special attention to handle the potential conflict from incorporating multiple libraries in a job.

In order to bypass the FUSE overheads and ease the development complexity, Direct-FUSE is designed to provide multiple backend services for one application without crossing kernel boundary. It is built on top of *libsysio*, which is discussed in Section 2.1.2.

3 DESIGN AND IMPLEMENTATION

Our *Direct-FUSE* offers a unified API and is designed to provide multiple backend services for a single job while bypassing the FUSE kernel.

3.1 The Software Architecture of Direct-FUSE

Direct-FUSE mainly consists of three layers, i.e., *adapted-libsysio*, *lightweight-libfuse*, and *backend services*, shown in Fig. 4.

The adapted-libsysio is adapted from the original libsysio to support multiple distinct FUSE file system operations. We enable a differentiation method in adapted-libsysio to identify various backend services for applications and intercept its inode operations to underlay lightweight-libfuse. To better improve the performance, we allow a new I/O routine for data related operations. The original libsysio maintains complicated procedures for data operations. If the input parameters are not related to *iovec* structures, it first constructs an I/O context by the input parameters, then reads or writes data by the newly created I/O vectors. However, as most of FUSE file systems are built on top of FUSE high-level API, whose input parameters are similar to the general read() and write() of POSIX library, constructing I/O vectors and reversing them back to the original input parameters can largely degrade the I/O performance. Thus, we bypass the redundant steps in the original libsysio to further enhance the performance. But we still keep the permission checking in I/O path, which determines if a file is readable or writable in Direct-FUSE. For other operations maintained in Direct-FUSE, like metadata operations, they still go through the original libsysio code path with only a few modifications for backends differentiation.

Unlike the original libfuse that has both file system and block devices registrations, our lightweight-libfuse exposes unified file system call interfaces to distinguish underlying backend services. To support FUSE high-level API and work with adapted-libsysio, our lightweight-libfuse manipulates the *inode_op* from the original libsysio. *Inode_op* is similar to the fuse_operations of FUSE implementation, which contains a few file system and inode related operations. We re-define these operations to better fit our purpose and offer an abstract interface to file system operations.

The backend services are operations of multiple user-level file systems, serving different kinds of data. For example, we can enable both direct access to remote volumes through SSHFS and the direct communication from application to GlusterFS daemons. All backend services are built with customized file system operations defined with FUSE high-level API. The customized operations (file system operations defined via FUSE high-level API) in fuse_operations of each backend service are indicated during the backend initialization.

Different from the original FUSE file system shown in Fig. 1, all three layers are kept in user space, which helps Direct-FUSE to avoid the inter-process communication of the original FUSE file system resulted from the use of FUSE kernel module.

3.2 Differentiation of Multiple Backend Services

As mentioned, to avoid the kernel overheads of FUSE, some FUSE file systems provide libraries to allow the operations of user-defined file systems to be directly accessed by applications. Because different kinds of data are handled by different file systems, users have to manage multiple file paths and file descriptors of various file systems with special cares. To mitigate the difficulty and amount of work when accessing various backend file systems in one job, a differentiation method is needed. Direct-FUSE handles the file path and file descriptor operations differently.

3.2.1 *File Path Operations.*

For file path operations, Direct-FUSE appends the file path after the desired file system prefix and passes them as a single parameter. When Direct-FUSE intercepts any file path operations, it checks the existence of the prefix and the path in mount list, which records information of mounted backends. If both the path and the prefix are found, Direct-FUSE returns the mount point information including a pointer to the underlying backend service's operations via lightweight-libfuse. The reason for incorporating both file path and file system prefix is that the same file system may be mounted on two different mount points. Simply applying file path may not be sufficient when this problem occurs.

We take open() as an example. When an open() is issued from a program, Direct-FUSE first verifies the indicated path and prefix, then dispatches a request to the corresponding backend. During the path and the prefix verification, Direct-FUSE compares the input argument with mounted backends' paths and prefixes. Once the matched path and prefix are found, a pnode that contains a chain of pointers to that backend is retrieved or created based on need. After that, the open() is directed to the user-defined open operation by chained pointers of pnode which goes through adapted-libsysio and lightweight-libfuse to reach backend services. Once the operation is completed, an open file record is created or initialized, and a free file descriptor is also assigned to the file record and returned back to the application program.

3.2.2 *File Descriptor Operations.*

When managing file descriptors, we don't simply apply ranges to file descriptors of different backends (i.e., file descriptors from different incorporated backends are assigned to different ranges). When a file is opened, a record has to be carried through to trace the status of the file. However, if the range policy is applied, an additional mapping is also needed to map file descriptors to opened file records.

Direct-FUSE returns the index of file records in open file table as the file descriptor. File records not only trace file status, such as current stream position of a file, but also coordinate open files with corresponding backends by storing pnode inside records. Once a file record is retrieved through its file descriptor, any backend services can be easily reached to invoke any related file system calls.

We also use an example here, i.e. write(), for illustration. A file descriptor is associated with an open file record. Similar to the open(), once a file record is retrieved from a given file descriptor, the corresponding backend services are found through the pnode in file record, and the correct write() function can be issued by the indicated backend service. When the write() function is completed, we update the file size and current stream position in the file record accordingly.

3.3 Implementation Issues

Although adding more backends to Direct-FUSE can further improve the flexibility for users to handle data, there are several requirements in achieving so: 1) the user-defined file system calls are implemented with FUSE high-level APIs, 2) we need an independent file system library, and 3) the backend is implemented in C/C++ or has to be binary compatible with C/C++. We will elaborate on each requirement next.

Firstly, similar to Direct-FUSE which interacts with backend services through FUSE high-level API, a new backend needs to do the same. FUSE file systems leverage either low-level (e.g. MooseFS) or high-level (e.g. SSHFS) FUSE API to complete user-defined file system calls. Both low-level and high-level API transfer incoming requests from the kernel to the main program through callbacks. When using low-level API, callbacks work with inodes. Thus, developers have to take care of path-to-inode mapping and also any associated cache for translation. In addition, several inode related operations (e.g., forget an inode, file lookup) also have to be specified by developers. However, these inode related operations will incur more user-kernel switches on metadata operations, leading to more performance lose. For example, when an open() is called in an application for an existing file, besides open(), a developer implemented lookup() is also triggered in FUSE file system process which requires crossing the kernel boundary. As mentioned in Section 2.3, often crossing the kernel boundary results in performance loss. Unlike the low-level API, callbacks in the high-level API directly interact with file names and paths instead of inodes. There is no additional development need on path-to-inode mapping, and no extra kernel boundary crossing in file system calls. Therefore, we choose to leverage FUSE high-level API in our Direct-FUSE.

Secondly, because all backend services are used as user-level libraries, for the FUSE file systems without independent libraries to bypass the FUSE kernel, users have to isolate the libraries by themselves. Such isolation involves modifying the original initialization function of FUSE file system clients, and defining globalized FUSE operations and unmount function. In detail, users have to modify the initialization function (*main()* of FUSE file system client) to preclude the *fuse_mount()* function as the fuse_mount() builds the communication channel between user-level processes and the FUSE kernel module, which is avoided in Direct-FUSE. However, since fuse_mount() is omitted in the new initialization function, it eliminates the ability to pass the special purpose data to FUSE backends. Although not every FUSE backend needs the transited data, if passing data is inevitable, the data has to be globalized in Direct-FUSE when creating the library.

Finally, because Direct-FUSE is utilized as a user-level library, backend file systems have to be implemented in the same language with Direct-FUSE or be binary compatible with the implementation language (C/C++ in our case). For instance, with Rust's Foreign Function Interface [2], using the mixture of C/C++ with Rust is not a problem.

4 EXPERIMENTAL EVALUATION

In this section, we present I/O bandwidth of Direct-FUSE for local and distributed file systems, and also an overhead analysis for Direct-FUSE.

4.1 Experimentation Environment

Our experiments are conducted on an in-house cluster, called Innovation. Each machine is equipped with 10 dual-socket Intel Xeon(R) CPU E5-2650 cores, 64 GB memory, and a 1 TB Seagate ST91000640NS SATA disk.

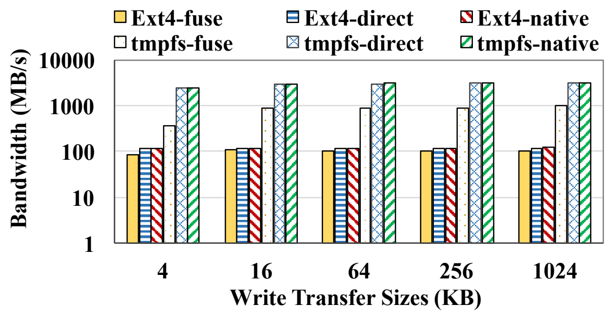

Figure 5: Sequential write bandwidth measured via Iozone.

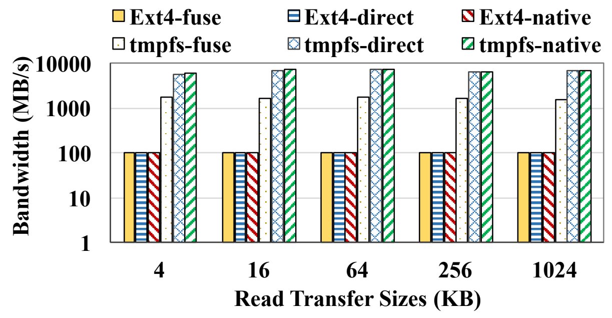

Figure 6: Sequential read bandwidth measured via Iozone.

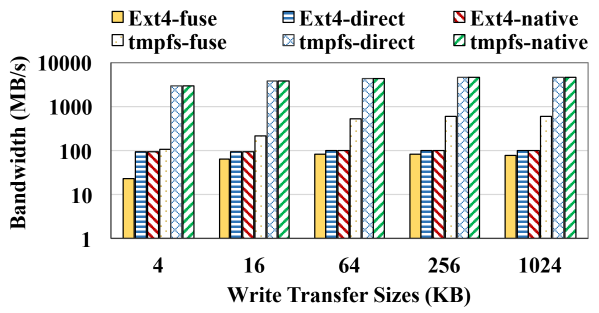

Figure 7: Random write bandwidth measured via Iozone.

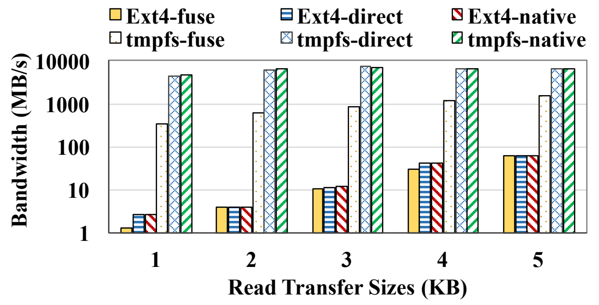

Figure 8: Random read bandwidth measured via Iozone.

4.2 The I/O Bandwidth of Direct-FUSE on Local File System

In this section, we compare the I/O performance of a FUSE file system, our Direct-FUSE, and a native file system on different storage media. Ext4 and tmpfs are used as the underlying file system on disk and memory, correspondingly. As shown in Fig. 5, *Ext4-fuse* is a FUSE file system on top of the native Ext4 file system. *Ext4-direct* is our Direct-FUSE on top of Ext4. *Ext4-native* is the original Ext4. Similarly, *tmpfs-fuse*, *tmpfs-direct*, and *tmpfs-native* denote a FUSE file system on tmpfs, Direct-FUSE on tmpfs and the original tmpfs, respectively.

The sequential read/write bandwidth in this section is measured by Iozone benchmark [17]. In all tests, the size of the read/write file is 1 GB, and the transfer sizes range from 4 KB to 1 MB. All experiments were repeated at least five times until results became stable. The reported numbers are the average of all runs. To get better performance from the FUSE file system, we enable *big_write*, *max_read*, *direct_io*, and *max_readahead* to reduce the number of exchanged FUSE requests between the test process and the file system process.

4.2.1 Sequential Read & Write. Fig. 5 shows that, when the transfer size is small, the bandwidth of *Ext4-fuse* is lower than the *Ext4-direct*, e.g. 35.6% for 4 KB. Although we enable *max_write* to shrink I/O requests between the FUSE kernel module and the process, incoming 4 KB data segments do not benefit from larger data chunk size, since every issued write only sends 4 KB data. On average, *Ext4-direct* outperforms *Ext4-fuse* by 11.9%. In addition, our design only loses at most 2.5% of I/O bandwidth when compared with *Ext4-native*. We also see a similar trend of write bandwidth on tmpfs tests, but with a more obvious performance enhancement. Because tmpfs utilizes a portion of the memory as storage media which has lower latency, it is more sensitive to the FUSE overhead. As a result, the overhead (at most 5.7%) of Direct-FUSE on memory is slightly higher than the overhead on disk. The detailed overhead analysis of Direct-FUSE will be discussed in Section 4.4.

In addition, Fig. 6 shows the read performance. We can see that the sequential read bandwidth of *Ext4-fuse*, *Ext4-direct*, and *Ext4-native* are very close. This is mainly due to the read related mount options as mentioned earlier, which enables the prefetching (*max_readahead*), fewer FUSE requests (*max_read*), and direct data transfer (*direct_io*) between the test process and the file system process in reading. Similar to the write performance shown in Fig. 5, the read performance of *tmpfs-fuse* is lower than *tmpfs-direct* and *tmpfs-native* by at least 226% and 243%, and up to 334% and 355%, respectively.

Overall, our Direct-FUSE delivers higher I/O bandwidth compared to the FUSE-native file systems on both disk and memory. It also demonstrates that our Direct-FUSE only introduce trivial overheads.

4.2.2 Random Read & Write. The bandwidths of random write, shown in Fig. 7, of the native file system, FUSE file system, and Direct-FUSE on disk and memory have a similar trend shown in Fig. 5. *Ext4-direct* exceeds *Ext4-fuse* by at least 16.73% when the transfer size equals to 4 MB, and up to 3.04x when transfer size is 4 KB. The bandwidth of *tmpfs-direct* outperforms *tmpfs-fuse* at least 5.7x and upto 25.2x. *Ext4-direct* and *tmpfs-direct* import less than 2% of additional overhead compared with *Ext4-native* and *tmpfs-native*.

In Fig, 8, the random read bandwidths of all tests are similar to Fig. 6. The overall trend of the read bandwidth of different file systems on the two storage media is also similar to Fig. 6. *Ext4-fuse*

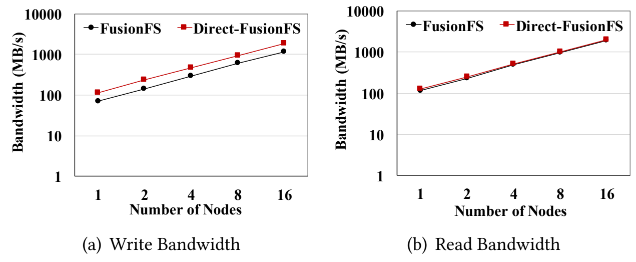

Figure 9: I/O bandwidth of FusionFS and Direct-FUSE.

and *Ext4-direct* have almost the same bandwidth, and *tmpfs-direct* delivers at least 68.95% higher performance than *tmpfs-fuse*. And the additional overheads in both *Ext4-direct* and *tmpfs-disk* occupy less than 3% of raw bandwidth.

4.3 The I/O Bandwidth of Direct-FUSE on Distributed File System

To test I/O performance on distributed FUSE file systems, we enable FusionFS user-level library as one of our distributed backends (Direct-FusionFS) and also compare with the FUSE-based FusionFS (FusionFS). We write 1 GB file on each node with 1 MB transfer size. As we can see from Fig. 9(a) and Fig. 9(b), doubling the number of nodes yields doubled throughput for both read and write, which demonstrates the linear scalability of Direct-FusionFS.

Similar to the results in Section 4.2, the write bandwidth benefits more on the FUSE kernel bypassing, and the read delivers almost the same performance trend as the original FusionFS. As we mount original FusionFS with tuned performance, the read favors from *max_read*, *max_readahead*, and *direct_io*. On average, Direct-FusionFS outperforms FusionFS by 38.35% on write and 6.3% on read. In addition, if further increasing the transfer size, the similar performance trend can still be observed since the performance benefits come from the FUSE kernel bypassing in Direct-FUSE.

4.4 The Overheads in Direct-FUSE

Considering that memory is more sensitive to the overheads, we conduct the overheads analysis on tmpfs.

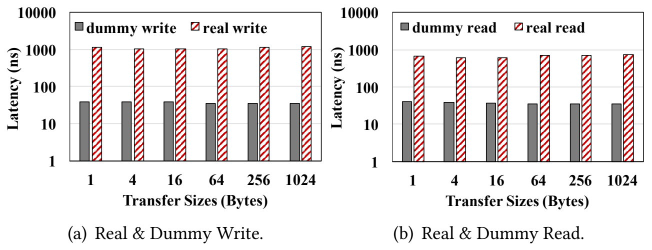

Figure 10: Breakdown analysis of W/R in Direct-FUSE.

4.4.1 *Cost of I/O Processing in Direct-FUSE.*

To demonstrate the performance improvement mentioned in Section 4.2 and Section 4.3, we use dummy read/write operations of Direct-FUSE in our experiments. The dummy operations go through the entire code path of the functions but return immediately to the front end once reaching underlying file systems. Therefore, no data is actually written or read by backends. The latency of dummy operations represents the overhead introduced by Direct-FUSE since our Direct-FUSE also maintains a few metadata operations for backends differentiation.

As shown in Fig. 10(a) and Fig. 10(b), the dummy read/write only takes about 38 ns, which is less than 3% of a complete read/write function time in Direct-FUSE. This indicates that, in spite of memory being sensitive to the overheads, the extra latency caused by Direct-FUSE is still negligible.

Table 2: Number of context switches for I/O in tmpfs-fuse and tmpfs-direct.

	tmpfs-fuse		tmpfs-direct	
Transfer Size (KB)	Write	Read	Write	Read
4	1000	1000	0	0
16	1000	1000	0	0
64	1000	1000	0	0
128	1000	1000	0	0
256	2000	2000	0	0
1024	8000	8000	0	0

4.4.2 *Number of Context Switches for Direct-FUSE.*

For the context switches analysis, we use the Linux *perf_event* library to monitor context switch events of particular Direct-FUSE APIs on the application side.

We use the same FUSE mount setting as in Section 4.2. 1000 I/O requests are issued to the *tmpfs-fuse* and *tmpfs-direct* in the tests. As shown in Table. 2, when the transfer size is less or equal to 128 KB, only one FUSE request is sent to the user-level file system process by the kernel module for each write; when the transfer size is greater than 128 KB, the large I/O data chunk is split into multiple 128 KB data chunks to be sent to the process. Conversely, as no context switch is generated in Direct-FUSE, there is no such problem when issuing read and write requests. Therefore, the expensive FUSE overheads have been reduced. These results demonstrate the performance improvements in Section 4.2 and 4.3.

5 RELATED WORK

The FUSE overheads are a notable concern for lots of system developers and researchers. Some researches are completed on analyzing FUSE-based file system performance. Rajgarhia et al. [20] implemented a JAVA-based FUSE API, which uses the Java Native Interface (JNI) to communicate between the C and Java layers. They showed the performance difference the JAVA-based and original C-based implementations. However, they only provided limited FUSE overheads analysis. Tarasov et al. [26] also utilized a FUSE file system overlaying Ext4 (FUSE-Ext4). Although they presented lots of file access patterns with FUSE-Ext4 on different storage devices, they didn't show cost breakdown for metadata and data operations in FUSE file system. Vangoor et al. [28] developed a FUSE-based stackable passthrough file system to investigate FUSE overheads. They mainly focused on instrumenting FUSE to extract useful statistics and traces to analyze performance bottlenecks.

Some methods in [9, 10, 14, 16, 21, 23, 32] are applied to improve the FUSE performance. File systems such as TableFS [21], FusionFS [32], OrangeFS [14], GlusterFS [9], and Gfarm [27] enables a distinct library to avoid crossing kernel boundary. However, when multiple file systems are needed for special purpose data, simply utilizing them in a job may improve the program complexity as an additional distinguishing method is necessary. Narayan et al. [16] proposed to marry in kernel-stackable file system [31] with FUSE. They combined ATTEST [15] with their design to provide a filter to files so that only specific extended attributes are exported to the user-space FUSE file system process. Shun et al. [10] allowed direct I/O access to local storage without routing requests through FUSE file system process in FUSE-based distributed file systems. As the design is mainly built for Gfarm, non-trivial efforts may be needed to integrate this method with other FUSE-based distributed file systems. Although some non-FUSE user-level file systems leverage LD_PRELOAD and linker-assisted wrapper functions to intercept file system calls, such as CRUISE [18] and BurstFS [29], some file system calls cannot be well covered by the dynamic link. However, this can be easily addressed in Direct-FUSE.

There are also some I/O forwarding infrastructures [5, 6, 11]. Iskra et al. [11] leveraged their design as I/O daemon implemented for compute node as file system clients. However, the I/O daemon leads to additional context switches between the application process and itself. And as Ali et al. [6] built their work over [11], their design also suffers the cost of context switches. libsysio [5] is introduced as a library, but it is limited to lustre file system.

6 CONCLUSION

Due to the complexities associated with developing kernel-level file systems, special-purpose file systems for particular I/O workloads are developed as user-level file systems, commonly using the FUSE framework. In this paper, we analyzed the FUSE overheads in FUSE file system calls, showed how I/O bandwidth was affected by the overheads, and broke down the cost of both metadata and data operations. To address the overheads associated with FUSE, we developed Direct-FUSE, a framework that supports multiple FUSE or other special-purpose user-level file systems in a single job. Because Direct-FUSE removes the need to cross the boundary between user and kernel space for I/O calls, it can significantly improve the I/O performance of FUSE file systems. Our experiments show that compared with original FUSE file system over some native file systems (e.g, Ext4, tmpfs), our Direct-FUSE can improve performance by 11.9% on average.

Acknowledgment

This work is performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-744728.

REFERENCES

[1] [n. d.]. Catalyst. http://computation.llnl.gov/computers/catalyst.
[2] [n. d.]. Rust's Foreign Function Interface. https://doc.rust-lang.org/book/first-edition/ffi.html.
[3] [n. d.]. Sierra. https://computation.llnl.gov/computers/sierra.
[4] [n. d.]. Summit. https://www.olcf.ornl.gov/summit/.
[5] [n. d.]. The SYSIO library. https://sourceforge.net/projects/libsysio/.
[6] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and Ponnuswamy Sadayappan. 2009. Scalable I/O Forwarding Framework for High-performance Computing Systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on.* IEEE, 1–10.
[7] Arthur S Bland, Jack C Wells, Otis E Messer, Oscar R Hernandez, and James H Rogers. 2012. Titan: Early Experience With The Cray XK6 at Oak Ridge National Laboratory. In *Proceedings of cray user group conference (CUG 2012)*. Cray User Group Stuttgart, Germany, 3–4.
[8] Arnaldo Carvalho de Melo. 2010. The New LinuxâĂŹperfâĂŹTools. In *Slides from Linux Kongress*, Vol. 18.
[9] Red Hat. 2012. GlusterFS.
[10] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. 2012. Optimizing local file accesses for FUSE-based distributed storage. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 760–765.
[11] Kamil Iskra, John W Romein, Kazutomo Yoshii, and Pete Beckman. 2008. ZOID: I/O-forwarding Infrastructure for Petascale Architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 153–162.
[12] R Jones. 2007. Gmail Filesystem-GmailFS.
[13] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying The Cost of Context Switch. In *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2.
[14] Micheal Moore, David Bonnie, Becky Ligon, Mike Marshall, Walt Ligon, Nicholas Mills, Elaine Quarles, Sam Sampson, Shuangyang Yang, and Boyd Wilson. 2011. OrangeFS: Advancing PVFS. *FAST Poster Session* (2011).
[15] Sumit Narayan and John A Chandy. 2010. Attest: Attributes-based Extendable Storage. *Journal of Systems and Software* 83, 4 (2010), 548–556.
[16] Sumit Narayan, Rohit K Mehta, and John A Chandy. 2010. User Space Storage System Stack Modules with File Level Control. In *Proceedings of the 12th Annual Linux Symposium in Ottawa*. 189–196.
[17] William D Norcott and Don Capps. 2003. Iozone Filesystem Benchmark.
[18] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K Panda. 2013. 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*. ACM, 143–154.
[19] Aditya Rajgarhia and Ashish Gehani. 2010. Performance and Extension of User Space File Systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 206–213.
[20] Aditya Rajgarhia and Ashish Gehani. 2010. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 206–213.
[21] Kai Ren and Garth A Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System.. In *USENIX Annual Technical Conference*. 145–156.
[22] Peter Snyder. 1990. tmpfs: A Virtual Memory File System. In *Proceedings of the autumn 1990 EUUG Conference*. 241–248.
[23] Junsup Song and Dongkun Shin. 2014. Performance Improvement With Zero Copy Technique on FUSE-based Consumer Devices. In *Consumer Electronics (ICCE), 2014 IEEE International Conference on*. IEEE, 434–435.
[24] Miklos Szeredi. 2004. SSHFS: SSH Filesystem.
[25] Miklos Szeredi et al. 2010. Fuse: Filesystem in Userspace. *Accessed on* (2010).
[26] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. 2015. Terra Incognita: On the Practicality of User-Space File Systems.. In *HotStorage*.
[27] Osamu Tatebe, Kohei Hiraga, and Noriyuki Soda. 2010. Gfarm Grid File System. *New Generation Computing* 28, 3 (2010), 257–275.
[28] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems.. In *FAST*. 59–72.
[29] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. 2016. An Ephemeral Burst-buffer File System for Scientific Applications. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 807–818.
[30] Teng Wang, Adam Moody, Yue Zhu, Kathryn Mohror, Kento Sato, Tanzima Islam, and Weikuan Yu. 2017. Metakv: A Key-Value Store for Metadata Management of Distributed Burst Buffers. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 1174–1183.
[31] Erez Zadok and Jason Nieh. 2000. FiST: A Language Stackable File Systems. (2000).
[32] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. 2014. Fusionfs: Toward Supporting Data-intensive Scientific Applications on Extreme-scale High-performance Computing Systems. In *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 61–70.
[33] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the 10th Parallel Data Storage Workshop*. ACM, 1–6.