IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 3, 2019, accepted September 18, 2019, date of publication September 23, 2019,
date of current version October 4, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2942954

Defuse: Decoupling Metadata and Data Processing in FUSE Framework for Performance Improvement

WENRUI YAN[1], JIE YAO[2], (Member, IEEE), AND QIANG CAO[1], (Senior Member, IEEE)
[1]Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System of Ministry of Education, Wuhan 430074, China
[2]School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

Corresponding authors: Jie Yao (jackyao@hust.edu.cn) and Qiang Cao (caoqiang@hust.edu.cn)

This work was supported in part by the Creative Research Group Project of NSFC under Grant 61821003, in part by the NSFC under Grant 61872156, in part by the National Key Research and Development Program of China under Grant 2018YFA0701804, in part by the Fundamental Research Funds for the Central Universities under Grant 2018KFYXKJC037, and in part by the Alibaba Group through Alibaba Innovative Research (AIR) Program.

ABSTRACT A popular user-space file system framework, FUSE, has been widely used for building various *customized file systems*(cFS) on top of the underlying *kernel file system*(kFS). A FUSE-based cFS gains adequate flexibility by developing its specific functions in user space, but brings extra user-kernel mode switches in the request processing flow owing to forwarding all requests from the FUSE kernel driver to the user-space daemon, thus degrading the overall performance. We observe that a file data request does not need to forward to the user-space daemon when its *file-to-file* mapping between the FUSE-based cFS and the underlying kFS remains unchanged. Based on the insight, we propose a modified FUSE framework - DeFUSE that decouples the processing flow of the metadata and data requests. The metadata requests still follow the original flow to reserve flexibility while the data requests are directly executed in the DeFUSE kernel driver maintaining the *file-to-file* mappings in the kernel, effectively eliminating the unnecessary mode switches. We have implemented the DeFUSE framework and ported three representative FUSE-based cFSs to DeFUSE. The result shows that for data-centric workloads, the throughput of DeFUSE-based cFSs increases up to 3.5X for write and 3.8X for read respectively, compared to their corresponding FUSE-based implementations. DeFUSE is available on Github.

INDEX TERMS FUSE, file system, file mapping, performance optimization.

I. INTRODUCTION

File systems are ubiquitously deployed in a myriad of storage devices and large-scale systems. They provide a common programmable interface (i.e. POSIX) for applications to access file data by implementing file operations upon underlying storage devices. Many traditional file systems are implemented in the OS kernel, such as Ext4 [1], f2fs [2], and ZFS [3], to directly maintain file-to-storage layout and to execute IO processes in OS kernel. The kernel file systems efficiently make full use of IO characteristics of the storage media while experiencing complex and lengthy development, which are error-prone with serious consequence of machine crash [4], [5].

The associate editor coordinating the review of this manuscript and approving it for publication was Huaqing Li.

Recently, user-space file systems rose in popularity thank to quick prototyping of new approaches and functionalities. The most popular user-space file system framework is FUSE (File system in user-space) [6]. For file system developers, a typical use case of FUSE framework is to build the *customized file system* (referred as **cFS** hereafter in this paper) stacking on top of mature *kernel file system* (referred as **kFS** hereafter in this paper). Rich user-space libraries can help extend the feature of the kFS (e.g. compression and self-defined encoding). Meanwhile, the user-space codes are more convenient to debug and test than kernel codes. As a consequence, FUSE has become a popular framework adopted by many existing file systems such as Ceph client [7] and Hadoop client [8], the client of cloud storage system Google Cloud Storage [9]; file system in data migration node MarFS [10]; a composited file system [11], a tiered optical storage system ROS [12]. These file systems can support

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

scalable big-data analysis and machine-learning applications such as Hadoop [13], TensorFlow [14], and SPARK [15], using legacy programmable interface (i.e. POSIX). Some of them have been deployed in production environments.

For the FUSE-based cFSs which are stacking on kFSs, their specific global namespaces are constructed by designing the specific *directory-to-directory* and *file-to-file* mapping between the cFSs and kFSs. A cFS generally leaves the complex and error-prone *file-to-storage* mapping to the underlying kFS. For *directory-to-directory* mapping, one or more individual namespaces of kFSs can be combined together or remapped to build the global namespace of cFSs. For *file-to-file* mapping, multiple files in a cFS can be aggregated into one file in the kFS [11] or one file in a cFS can be split into several files in the kFS [12].

Unfortunately, FUSE-based cFSs reap flexibility by sacrificing performance in both latency and throughput. A FUSE-based cFS only achieves about 37% the performance that its kFS provides [16]. The root cause of such performance degradation is that IO routine in the FUSE-based cFS leads to extra *one* or more user-kernel-user mode switch compared to the kFS. Note that the mapping relationship between a FUSE-based cFS and its kFS is generally defined, processed, and stored by the FUSE user-space file system daemon. For a FUSE-based cFS, all requests first trap into the FUSE kernel driver, and then forward to the FUSE user-space daemon to be processed with customized logic, leading to the extra user-kernel-user mode switches. More specifically, we further divide the requests of the FUSE-based cFS into three classes: the directory, file metadata and file data requests. The directory operation creates, modifies or obtains the *directory-to-directory* mapping of a cFS directory; The file metadata request defines or modifies the *file-to-file* mapping of a cFS file; The file data request accesses the data on the corresponding one or multiple files in the kFS. The user-kernel-user mode switches in the directory and file metadata request are essential because the mapping relationship must be manipulated in user space to gain the flexibility of FUSE. However, when merely accessing the file data, the file data requests actually do not modify its relevant *file-to-file* mapping. Therefore, when the *file-to-file* mapping is fixed and can be kept in the kernel, the file data requests do not need to be forwarded into the FUSE user-space daemon so that the extra user-kernel-user mode switch in the file data requests can be eliminated effectively.

This observation inspires us to present a novel framework - DeFUSE, as shown in Figure 1, to reduce unnecessary user-kernel-user mode switches during accessing file data by directly executing data I/Os within the kernel file system.

The main contribution of this paper is summarized as follows:

- We propose a DeFUSE based on FUSE framework to decouple the processing flow of the metadata and data request. The directory and file metadata requests follow the original flow of FUSE to reserve flexibility. The processing flow of the file data requests bypasses the FUSE users-space file system daemon and completely executes in the kernel, avoiding unnecessary mode switches.
- We generalize the *file-to-file* mapping of FUSE-based cFSs into three fundamental patterns. We design a file mapping structure *Data_Map* to uniformly describe such three *file-to-file* mapping patterns. DeFUSE kernel driver maintains and uses *Data_Map*s that are delivered from the user-space daemon as requested.
- We implement the DeFUSE framework by simply modifying the FUSE user-space daemon and the kernel driver. We also transplant three representative FUSE-based cFSs to DeFUSE-based cFSs.
- We evaluate the performance of these DeFUSE-based cFSs and compare them with their corresponding FUSE-based implementations. The result shows that for data-centric workloads, the throughput of these file systems has significantly improved up to 3.8X for read and 3.5X for write respectively.

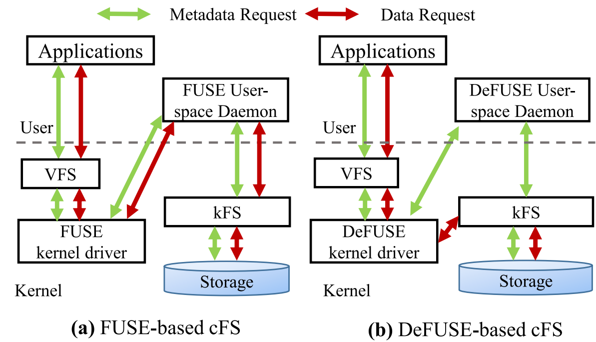


FIGURE 1. The architecture of FUSE and DeFUSE framework.

The rest of the paper is organized as follows: Section II describes FUSE framework and the design patterns of FUSE-based cFS. We present the motivation of this work. Section III discusses the design and implementation details of DeFUSE framework. Section IV gives the performance evaluation of the DeFUSE framework. Section V presents current researches on the optimization of FUSE and FUSE-based cFSs. Section VI concludes the paper.

II. BACKGROUND AND MOTIVATION

FUSE (File system in User Space) offers a generic file system framework for developing cFSs stacking atop of kFSs. There have been more than 100 FUSE-based file system available on Github [17]. In this section, we first describe the architecture of the FUSE framework and the design pattern of current FUSE-based cFSs. Then we analyze the mode switches in the FUSE-based cFSs. Lastly, we present the motivation of this work.

A. FUSE ARCHITECTURE

The architecture of FUSE framework is shown in Figure 2. It consists of two modules: **FUSE kernel module** and **User-space file system daemon**.

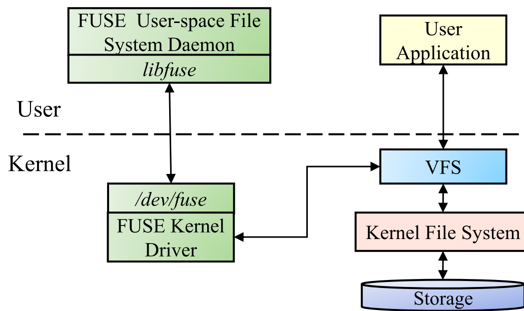


FIGURE 2. The architecture of FUSE framework.

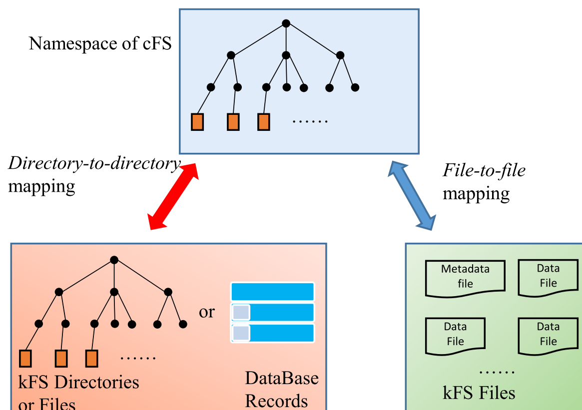


FIGURE 3. The mapping of the directory and file between the cFS and kFS.

FUSE Kernel Driver: The FUSE kernel driver registers a *fuse* file system driver in the Linux's VFS and registers a block device */dev/fuse*. The FUSE kernel driver forwards the requests to the device file */dev/fuse*. The device file */dev/fuse* is the bridge between the FUSE kernel driver and the user-space daemon. Both the user-space file system daemon and the FUSE kernel module can read/write data from/to the device file.

User-Space File System Daemon: The user-space file system daemon reads the requests from */dev/fuse*, translates the request, parses the request type and parameters, then calls the corresponding user-space APIs according to the request type. File system developers can implement their file system functions (e.g., compression or de-duplication) in the user-space APIs. When the corresponding user-space APIs have finished their functions, the user-space daemon writes back the result to FUSE kernel driver via */dev/fuse*.

B. FUSE-BASED FILE SYSTEMS DESIGN PATTERN

The critical work for a FUSE-based cFS is to consider the specific namespace construction between the cFS and its corresponding kFSs as shown in Figure 3.

For the directory mapping in FUSE-based cFSs, there are two typical ways: One way is to directly map kFS directories to the FUSE-based cFS based on a pre-defined rule or algorithms (e.g., adding node id or specific prefix string to the kFSąŕs directories) [12]; Another way is to indirectly store the namespace relationship in database or configuration files [10]. When the FUSE-based cFS executes the *mkdir* and *readdir* requests to a directory, the *directory-to-directory* mapping can be created and accessed accordingly. Besides of the mapping, the cFS metadata (e.g., the attribute of the file, such as the file size, owner and creation time) are conveniently extended, for example, the file data content abstract, access rate and attributes about the upper layer logic [10], [12]. These extended metadata can be used for specific processing and analysis.

For the file mapping in the FUSE-based cFS, the file data may be split or aggregated on several files in the kFS [11], [12]. The *file-to-file* mapping between a cFS file and its related kFS files also can be defined by a algorithms or a list stored in a configuration file. When the user-space *read* and *write* APIs are called to access file data, the *file-to-file* mapping relationship must be acquired by executing user-space *open* or *create* APIs. The *file-to-file* mapping remains unchanged during the file data requests, such as OLFS [12], PLFS [18].

C. FUSE USER-SPACE APIS

FUSE-based cFS developers can implement their file systems functions by adding codes in the user-space APIs. The FUSE framework provides two sets of user-space APIs: *high-level* API and *low-level* API. The developers can choose one set of the APIs to implement their cFS.

Low-level APIs process the request from */dev/fuse*. The prototype of *low-level* APIs are as follow:

```
low_level_fs_operations(struct req*
                              req,...)
```

The *struct req* is the data structure received from the kernel driver, it contains the request type and data location information. While the *high-level* APIs are on top of the *low-level* APIs. After the *low-level* APIs handle the *struct req* and transform the information to path-based, the *high-level* APIs can use the file path to deal with the mapping between the FUSE-based cFS and the kFS. The prototype of *high-level* APIs are as follow:

```
high_level_fs_operations(char * path,...)
```

The difference between *low-level* and *high-level* APIs is that file system developers are explicitly handling different request parameters in the former case. When developers use the *high-level* APIs, they merely process the path string mapping between a FUSE-based cFS and its kFS in an intuitive way. Therefore, most production FUSE-based cFSs employ*high-level* API mode. So far, there are over one hundred open source FUSE-based cFSs in Github, and over 90% of them are implemented with *high-level* APIs [17]. In the rest discussion of the paper, cFSs are implemented with *high-level* APIs by default.

Taking *high-level* APIs for example, a FUSE-based cFS needs to implement the functions listed in Table 1. We divide

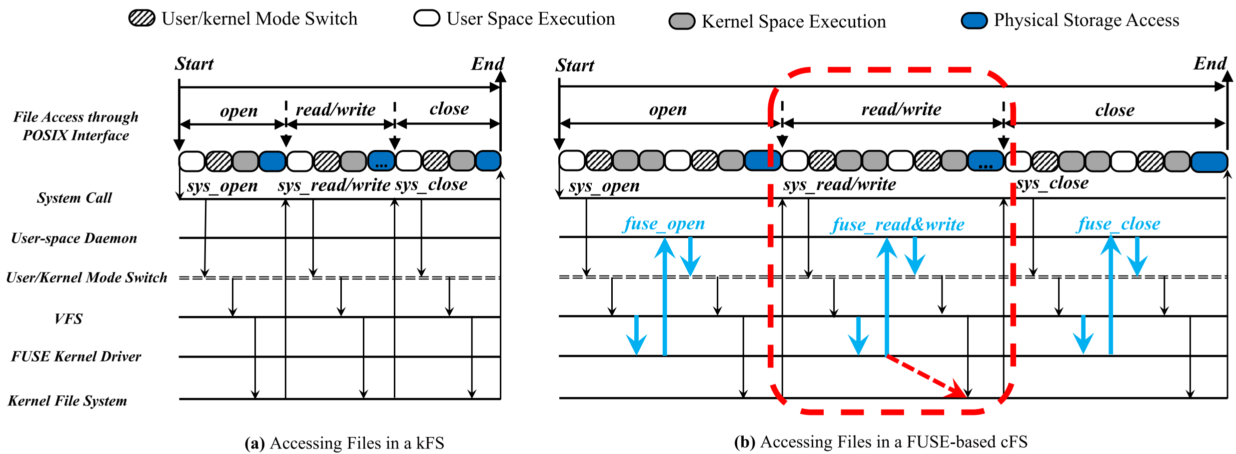


FIGURE 4. File access consists of three steps: *open*, *read&write* and *close*. When accessing a file in FUSE-based cFS, the blue arrows shows that each of these steps induces a user-kernel-user mode switch. When the file is large, the *read&write* may introduce large amount of user-kernel mode switches. The red dotted arrow in the *read&write* part indicates that the file data IO can be completely executed in the FUSE kernel driver.

TABLE 1. FUSE user-space APIs. We divide the user-space APIs into three categories: Directory, File Metadata and File Data.

API Type	Interface Functions
Directory	*readdir,mkdir*
File Metadata	*open,create,release*
File Data	*read,write*

the user-space APIs into three categories: the *Directory* APIs which are used to process the *directory-to-directory* mapping between cFS and kFS; the *File Metadata* APIs are used to deal with the *file-to-file* mapping between cFS and kFS; the *Data* APIs which are used to access the file data on the kFS.

D. MODE SWITCHES IN A FUSE-BASED CFS

As shown in Figure 4, when a user application issues a request to a file in the FUSE-based cFS, with the system calls such as *sys_open*, *sys_read*, *sys_write* and *sys_close*, the file access process traps into the kernel VFS module, and then the user application sleeps and waits for the result. Afterwards, VFS calls the registered FUSE kernel driver function and the FUSE kernel driver writes the request to */dev/fuse*; The user-space daemon reads the request from */dev/fuse*. It means the file access process switches back to user space. The user-space daemon translates the request, then performs the corresponding user-space API. Usually, the user-space API issues a request to the corresponding file(s) in the kFS, for example, *open* and *close* access the configuration file, *read* and *write* access the data file(s). Thus the file access process re-enters the kernel VFS module to actually access the kFS files. When the user-space API completes the request and obtains the result, the user-space daemon writes the result back to */dev/fuse*; FUSE kernel driver receives the result and wakes up the user application to finish the file access process.

From the comparison shown in Figure 4, it can be seen that the FUSE-based cFS introduces extra mode switches. And each call to the user-space API leads to *one* extra "round trip" mode switch from user space to kernel and then back to user space. When the user requests induce multiple file I/Os (up to 128KB per I/O), the processing flow of the request has to switch frequently between the user and kernel mode, thus resulting in severe throughput degradation. For example, when reading or writing a 1MB file with the IO size 4KB, there are *258* switches (one for the *open*, one for the *close*, and 256 for the *read/write*) user-kernel-user switches in a kFS. However, for a FUSE-based cFS, the count of the user-kernel-user switches is *516*, doubled compared to the kFS. Current FUSE framework supports an IO size up to *128KB*. Under *128KB* IO size the user-kernel mode switches count are *10* in the kFS and *20* in the cFS respectively. The user-kernel-user switch count in the cFS is reduced but still doubled compared to the kFS. The experimental result shows that in a FUSE-based cFS, the kernel/user mode switches can cause 63% throughput degradation compared to its underlying Ext4 [16].

E. MOTIVATION

All request-processing flows of a FUSE-based cFS follow the *user-kernel-user* routine. However, for file data requests, we found that the *file-to-file* mapping generally remains unchanged after it is initially determined by the *open* or *create* API. If the mapping can be obtained in the kernel, the processing flows of the subsequent *read* and *write* are not essential to be forwarded to the user-space daemon, actual data IOs can be completed in the kernel as the red dotted arrow in Figure 4. Therefore, for file data accessing, which is a common case, especially for large files, a large amount of mode switches originally caused by file data IOs can be eliminated effectively. Based on this insight, we present DeFUSE framework to improve file data IO performance. In the next section, we will give details of DeFUSE framework.

III. DEFUSE DESIGN AND IMPLEMENTATION

In this section, we first present the design of DeFUSE framework. Then we give the implementation details of the

DeFUSE framework. At last, we apply the DeFUSE framework to three representative cFSs.

A. DEFUSE OVERVIEW

To achieve the design goal of the DeFUSE framework, the details efforts we made are listed below:

(1)

1) We modify the FUSE kernel driver and */dev/fuse* to handle the metadata and data requests according to different processing routines;
2) We define a generalized data structure *Data_Map*, to describe the *file-to-file* mapping of each file between cFS and kFS;
3) We design *Data_Map* delivery mechanism between the user-space daemon and DeFUSE kernel driver at runtime to support concurrent and parallel access;

B. DECOUPLING PROCESSING FLOW

DeFUSE framework inherits the overall architecture from FUSE but modifies its kernel module. To distinguish metadata and data processing, we first category all requests into four types: directory requests as *mkdir* and *readdir*; file metadata requests as *open*, *create*, and *close*; file data requests as *read* and *write*; the other requests in FUSE. In the DeFUSE framework, the directory, file metadata, and the other requests obey the original *user-kernel-user* processing flow. The file metadata requests as *open*, *create*, and *close*, process the *file-to-file* mapping in user space. However, the data requests as *read* and *write*, are not forwarded longer to the user-space daemon. In contrast, they are completely handled in the kernel. DeFUSE first identifies the request type as file accessing requests and other metadata requests.

To decouple processing flow of the file metadata and data request, we first identify the corresponding functions in the FUSE kernel driver handling the *open*, *create*, *read*, *write* and *close* requests. Then, we modify their respective processing logic. The *create* and *open* requests generate the *file-to-file* mapping structure of the requested file as *Data_Map* in user-space. And then, the *Data_Map* is delivered into to the DeFUSE kernel driver. Afterwards, the file data requests can be transformed into file data I/Os executed by the kFSs. Separately processing and storing the metadata and data on Metadata and Storage servers respectively is a common idea in distributed file systems [13], [19], Chen Youxu et.al [20] present a metadata prefetch technique for distributed file system to reduce the amount of requests to the metadata node. However, DeFUSE does not reduce the amount of metadata and data requests, but reduces the user-kernel mode switches within data accessing.

C. FILE-TO-FILE DATA STRUCTURE

When directly accessing a file data in the kernel, the key point is to design a kernel data structure, *Data_Map* that records the *file-to-file* mapping of the requested cFS file. *Data_Map* should is defined by the user-space daemon and used in the kernel. Furthermore, emphData_Map does not affect the original concurrency and parallelism of FUSE framework.

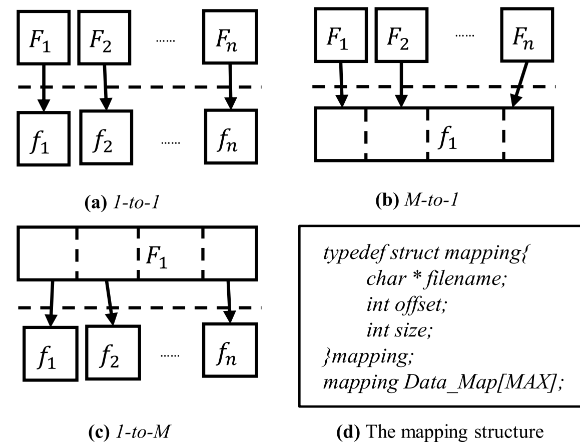


FIGURE 5. Three basic file mapping patterns between FUSE-based file system and underlying file system: *1-to-1*, *M-to-1* and *1-to-M*. The upper files in the FUSE-based file system actually are stored in its one or more corresponding physical files in the underlying file system.

TABLE 2. The file data mapping pattern of FUSE-based file systems. PLFS and OLFS have *1-to-1* and *1-to-M* pattern according to the configuration files.

File Data Mapping Pattern	The FUSE-based cFS Use Cases
1-to-1	OLFS [12] PLFS [18] fuse-dfs [13] SafeFS [21] Ceph Client [22] SSHFS [23] NTFS-3g [24] gcsfuse [9]
1-to-M	OLFS PLFS s3fs-fuse [25]
M-to-1	Composite File System] [11]

Each cFS file has its own *Data_Map* that uniquely identify this file within both the user-space daemon and the kernel. There are two kinds of file identifiers to represent a kFS file in the kernel: the file descriptor *fd* and the file path string *filename*. For file descriptor, each user-space process has its own file descriptor table while the OS kernel also maintains a global file descriptor table. Therefore, the *fd* of a file in user space and kernel space is not unique. However, the absolute file path of the kFS file is unique no matter in user space or in kernel space, so we choose the file path string *filename* as the unique identifier of a file in kFS.

Generally, *file-to-file* mapping combination between the cFS and kFS can be categorized into three basic types: *1-to-1* where a cFS file is completely stored in a single kFS file; *1-to-M* where a cFS file is split and stored in multiple kFS files; *M-to-1* where multiple cFS files are compacted into a single kFS file. The three kinds of file mapping are shown in Figure 5. We have made a brief classification for existing FUSE-based file systems, as shown in Table 2.

To abstract these three types of typical FUSE-based cFSs, the mapping structure *Data_Map* contains the kFS file path string *filename*, the *offset* in the kFS file and the *size* of file data in the kFS file. The offset parameter is for the *M-to-1* mapping, when many cFS files are compacted to a single

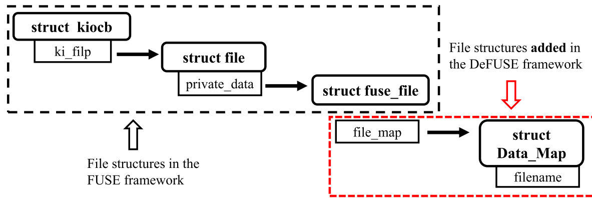


FIGURE 6. The *Data_Map* is added as a member of the structure *fuse_file*, inheriting the access control of the FUSE framework.

large kFS file, as shown in Figure 5, the location of the small file data requires the internal offset in the large file. When the file mapping is *1-to-1*, the *offset* is set to *zero*. When the mapping pattern are *1-to-M*, one cFS file is partitioned into multiple small kFS files so that there are multiple pairs of *filename*, *offset* and *size*. Hence the *Data_Map* is a structure array. DeFUSE supports *MAX* file partition number up to *16*, which is enough for current practical FUSE-based cFSs.

D. DATA_MAP DELIVERY

When the file metadata request passing in the user space daemon is completed, the callback function in the DeFUSE kernel driver can bring back a parameter structure of the user space. Therefore, we leverage a built-in tunnel in the callback function to effectively and efficiently deliver the *Data_Map* to the DeFUSE kernel driver.

When a *Data_Map* of a cFS file has been kept in the FUSE kernel driver, a *write* request could modify *size* and *offset* in the *Data_Map*. In this case, another application might be reading this file. Therefore, concurrent access to the *Data_Map* should be supported. Current FUSE framework supports the concurrent file access from multiple threads by managing the file structure *fuse_file* with a mutex_lock *fuse_mutex*. The design principle of DeFUSE is to make as least modification as possible to the FUSE framework. Therefore, instead of creating a new mutex_lock, we add the *Data_Map* as a member in the structure *fuse_file*, inheriting the existing concurrent access control in the FUSE framework, as shown in Figure 6.

A cFS file *Data_Map* lifecycle begins with the *open/create* request and ends with the *close* request. Therefore the mapping delivery between user-space and kernel may not significantly affect the throughput of DeFUSE-based cFSs when accessing large file, because the *Data_Map* only needs to be delivered twice in *open/create* and *close* requests. But when the user requests are small-file-dominated, the performance of the DeFUSE-based cFSs would degrade because of the frequent *Data_Map* delivery in the metadata processing. To accelerate the small files workload, we add a *M-Cache* queue to cache the used *Data_Map* in the kernel driver. The M-Cache queue is a hash table, the hash keys are the VFS *inode-id*s. Before the *open* request to a file is sent to user space, we check the file *inode-id* to find out whether the *Data_Map* is cached. If it has been cached, the *open* request does not need to be sent to the user space daemon. Otherwise, the *open* request follows the original processing flow, and the *Data_Map* will be added to the M-Cache queue. The M-Cache queue has a maximal length of 1024 with an LRU replacement approach. The performance impact of the *Data_map* delivery process is further evaluated in Section IV-D.

E. FILE REQUESTS HANDLING

In this section, we combine the processing flow of the metadata and data requests together, give a detailed description of how the file requests are served in a DeFUSE-based cFS. The processing flows of writing a new file and reading an existing file are respectively shown in Algorithm 1 and Algorithm 2.

Algorithm 1 Processing File Write Request

Require:
File request parameters, *FPath*;
File request parameters, *FData*;
The pre-defined mapping pattern, *Mp*;
Ensure:
The data file, *Df* ;
The metadata file, *Mf* ;
1: In *open* or *create* API, according to *Mp*, the *filename* in *Data_Map* are initiated with *FPath*, *size* and *offset* are set to zero, then the *Data_Map* is delivered to the DeFUSE kernel driver;
2: In DeFUSE kernel driver,*FData* are written to the specified file *Df* with the *filename*, then *size* and *offset* are updated;
3: In user-space *release* API, the *Data_Map* is delivered back, then create the metadata file *Mf* and write *Data_Map* to *Mf* ;
4: **return** *Df*, *Mf* ;

Algorithm 2 Processing File Read Request

Require:
The data file, *Df* ;
The metadata file, *Mf* ;
File request parameters, *FPath*;
File request parameters, *FSize*,*FOffset*;
Ensure:
File request parameters, *FData*;
1: In *open* API, according to *Mp*, the *Data_Map* are initiated by reading from the metadata file and delivered to DeFUSE kernel driver;
2: In DeFUSE kernel module, *FData* are read from *Df* with the parameters *filename*, *size* and *offset*;
3: In user-space *release* API, nothing needs to be done since *Data_Map* is not modified;
4: **return** *FData*;

As a result of Step 3, when writing a new file in a DeFUSE cFS, the other threads to access this file have to wait until the user-space *release* operation is finished until the *Data_Map* is written back to its corresponding metadata file, thus leading

TABLE 3. We present the code changes required to implement DeFUSE framework and port OLFS, fuse-dfs and PLFS.

Component	Lines of code	
	Original	Modified
libfuse	11K	10
FUSE kernel driver	12K	127
OLFS	13K	34
fuse-dfs	6K	42
PLFS	32K	34

```
1  static ssize_t fuse_file_read_iter(struct kiocb
     *iocb, struct iov_iter *to){
2  char * r_buf;
3  ssize_t r_len,s_len;
4  set_fs(KERNEL_DS);//Set the address state to
     kernel mode
5  //get the length and buf of the required data
6  r_len = iov_iter_count(to);
7  r_buf = (char *)vmalloc(r_len);
8  s_len = vfs_read(real_file,r_buf,r_len,&(iocb->
     ki_pos));
9  return s_len;
10 }
11
```

Listing 1. Part of the kernel read function.

to long access latency. The result of multiple threads to access files in the DeFUSE file system is illustrated in Section IV-C.

F. IMPLEMENTATION

The DeFUSE framework is implemented based on FUSE 3.0.0 in Linux 4.8.5. Table 3 shows the lines of code modification as we implement DeFUSE and port the FUSE-based cFSs. The detailed modification is listed below:

Libfuse: The common parameter transferred between DeFUSE kernel driver and the user-space daemon - struct *fuse_file_info* is defined in the header file *fuse.h*. Therefore, we add the mapping structure *Data_Map* as a member of the structure *fuse_file_info*.

FUSE Kernel Driver: In order to complete file data access in the DeFUSE kernel driver,, there are two methods: the *VFS-level* and the *Address Space* method. The *VFS-level* method is to replace the function *generic_read_iter* to VFS-level function *vfs_read*; The *Address Space* method is to replace the address space operation *fuse_readpages* with the *ext4_readpages*. Such two methods can serve the read request. The *Address Space* method is the typical implementation method of the kFS, it requires careful modification for the kFS module in code-level to export its static address space functions. However, when the kFS changes, this modified method will not work anymore. In contrast, the *VFS-level* method uses VFS-level function which is generic for all kinds of kFSs. Therefore, we modify the VFS-level function *vfs_read* and *vfs_write* to execute the file data access in the FUSE kernel driver as shown in Listing 1.

User-Space APIs: Because the file data operations are performed in the DeFUSE kernel driver, the user-space *read* and *write* APIs are bypassed. The user-space *open* and *create* APIs need to acquire the *file-to-file* mapping from the corresponding metadata file, database, or algorithms, then to

```
1  /* User space open function*/
2  static int fuse_open(const char *path, struct
     fuse_file_info *fi){
3  int res;
4  mapping Data_Map[MAX];
5  char data_path[MAX_LEN];
6  init_map(Data_Map);//initialize the mapping
     structure
7  data_path = getRealPath(path);//Get the actual
     data path
8  sprintf(Data_Map[0].filename,"%s",data_path);
9  return 0;
10 }
11
```

Listing 2. User space open function.

generate the *Data_Map* with the items of *filename*, *offset* and *size*, finally to assign the *Data_Map* to the member in the structure *fuse_file_info*. And the *release* function writes the *Data_Map* back to the metadata file or database. The contents of the *Data_Map* actually change according to the cFSąŕs functionality.

G. APPLICABILITY

To build a DeFUSE-based cFSs, it is needed to pre-define its featured *file-to-file* mapping. Similarly, for existing FUSE-based cFSs, in order to port them to DeFUSE, the key point is also to figure out the *file-to-file* mapping between their cFSs and kFSs, which is generally defined in *open* API or in the configuration files. To manifest applicability of DeFUSE, we choose three representative FUSE-based cFSs: OLFS [12], fuse-dfs [13] and PLFS [18], and port their implementations to DeFUSE framework. Note that these file systems have other rich features besides of their specific *file-to-file* mappings, but in this paper, we only focus on the *file-to-file* mapping part of these file systems. The following is the brief introduction of their functions and how we extract the *file-to-file* mappings of these file systems.

OLFS: OLFS is a file system on rack-based optical disc library, it provides a universal namespace for hard disks and thousands of optical discs. From function *olfs_open* in *olfs.c* in OLFS code, we can know that the underlying files are stored in the pre-allocated optical disc images, so the *filename* in *Data_Map* is the *disc-image-dir* plus the *path* in the *olfs_open* API.

Fuse-dfs: Fuse-dfs is a client file system of HDFS. It allows HDFS to be mounted as a common file system. Once fuse-dfs is mounted on the client side, the user can access files on the remote data nodes with POSIX file system APIs. The user's file data first is stored in the local temporary files, then the temporary files are transferred to the remote data nodes according to the file distribution rules from HDFS namenode in the background. The directory of the temporary files is configured in the configuration file *core-site.xml*, so the *filename* in *Data_Map* is the *tmp-dir* plus the *path* in the *dfs_open* API.

PLFS: PLFS is short for *Parallel Log-structured File System*. Parallel applications write their checkpoint data into a single shared file, leading to small and not aligned writes to the shared file, thus the performance is poor. PLFS remaps the

small writes into log structured writes to multiple shared files, greatly improving the performance for check-pointing. PLFS supports multiple work patterns: one shared checkpoint file for all applications or one checkpoint file for each application. We set the working pattern to the latter one and configure the data file directory to *data-dir* in the configuration file *.plfsrc*. The *filename* in *Data_Map* is the *data-dir* plus the *path* in the *plfs_open* API.

After we determine the *file-to-file* mapping of these cFSs, the *Data_Map* of these file systems can be acquired in the metadata request and sent to the DeFUSE kernel driver. Then the data request can be directly executed in the kernel, thus reducing the user-kernel mode switches. As shown in Table 3, the file system porting can be done with minor code changes.

IV. EVALUATION

In this section we validate correctness of porting DeFUSE-based cFSs, and then analyze the performance of the DeFUSE framework.

A. EVALUATION METHODOLOGY

To understand the applicability of the DeFUSE framework, we have ported three typical FUSE-based file systems to DeFUSE framework as mentioned in Section III-G. Then we give detailed discussion on how the DeFUSE framework can achieve performance improvement and how the extra *Data_Map* delivery affects the access latency of file system. Below are our evaluation metrics, workloads, and experiment environment.

DeFUSE Applicability Test: We choose three common FUSE-based cFSs: *OLFS*, *fuse-dfs* and *PLFS* and then transplant them to DeFUSE framework and mark them as: *De-OLFS*, *De-fuse-dfs* and *De-PLFS*. We run the *"single-stream-IO"* workload in Filebench [26], [27] to validate the applicability of the DeFUSE-based file system and the performance improvements compared to FUSE-based cFSs. The *"single-stream-IO"* workload reads or writes one 60 GB file with one thread, and the IO size is set to 1MB. The results report the file system throughput and access latency under the file system configurations.

DeFUSE Framework Analysis: After we validate the applicability of DeFUSE-based cFSs, in order to accurately analyze the impact of the mapping delivery process in DeFUSE, we implemented the *passthrough* file systems based on the FUSE and DeFUSE framework respectively, which simply pass all the cFS requests to the kFS with no extra processing in user-space APIs. These two file system configurations are marked as *FUSE FS* and *DeFUSE FS*. We further run the following data-centric workload and metadata-centric workload in Table 4 to understand how different workloads affect the performance of the DeFUSE-based cFS. The data-centric workloads work in both single thread mode and multiple thread mode. All workloads have four IO sizes of *4KB*, *32KB*, *128KB*, *1MB*. The File-server workload is metadata-centric, the workload concurrently accesses 200,000 files with multiple threads.

TABLE 4. The data-centric and metadata-centric workloads.

Workload Name	Workload Description
seq-rd-1th-1f	Single thread sequentially read from a single preallocated 60GB file.
seq-rd-32th-32f	32 threads sequentially read 32 preallocated 2GB files. Each thread reads its own file.
seq-wr-1th-1f	Single thread creates and sequentially writes a new 60GB file.
seq-wr-32th-32f	32 threads sequentially write 32 new 2GB files. Each thread writes its own file.
rnd-rd-Nth-1f	(1,32) threads randomly read a single preallocated 60GB file.
rnd-wr-Nth-1f	(1,32) thread randomly writes to a single preallocated 60GB file.
File-server-Mth	Consisting of small file open, read, write and append write. The IO size is 128KB and the append write size is 16KB. The numbers of files is scaled up to 200,000. The numbers of thread ranges in (1, 10, 20, 30, 40, 50).

TABLE 5. The experiment setup of the server.

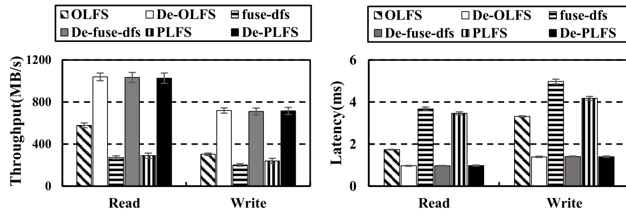
System version	Ubuntu Server 16.04
CPU	Intel Xeon CPU E5 14 Cores 2.0GHz
Memory	16GB
SSD	6 * 256GB, RAID-0

Experimental Setup: The performance of user-space file system depends on the underlying storage media and kernel file system. The hardware configurations of the server are shown in Table 5. The SSD arrays are configured as RAID-0 to get the full throughput of the storage media. The kFS we choose is Ext4 [1], for its stable performance, mature optimization, and prevalent deployment. The performance of *Ext4* works as the baseline. Each time before we run a workload, we clear the file system cache and remount the file system, to eliminate the effect of cache generated by the previous workload. All the workloads were running for more than 10 times to get the stable results. The final throughput and latency we reported in this section are the average values of 10 stable experiments results. The standard deviations of all these results range from 1.8% to 6.8%, considering that the standard deviation of SSD performance is around 6% [28], the experiment results can be considered as stable and and consistent.

B. DEFUSE APPLICABILITY

The throughputs under file system configurations are shown in Figure 7(a). And the amount of user-kernel mode switches are shown in Figure 8.

From the throughput result, we can see that the throughput of *De-OLFS*, *De-fuse-dfs* and *De-PLFS* respectively increased by 1.8X, 3.8X and 3.5X in read, as well as by 2.4X, 3.5X and 3.0X in write. These results show that the DeFUSE-based cFSs far outperform the FUSE-based file systems. The reason of the performance improvement is the reduced user-kernel mode switches as shown in Figure 8. The DeFUSE



(a) The throughput comparison of FUSE-based cFSs and DeFUSE-based cFSs.

(b) The latency comparison of FUSE-based cFSs and DeFUSE-based cFSs.

FIGURE 7. **The performance metrics of FUSE-based cFSs and DeFUSE-based cFSs. The results reported here are the average value of the repetitive experiments and the standard deviations of the results here range from 1.8% to 3.7%.**

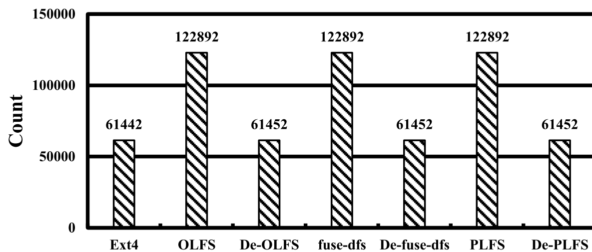


FIGURE 8. **The amount of user-kernel mode switches of FUSE-based cFSs and DeFUSE-based cFSs. The amount of Ext4 works as the baseline.**

cFSs only adds 10 extra user-kernel mode switches compared to Ext4 because the *getattr*, *open*, *release* requests in metadata processing. Compared to the FUSE cFSs, DeFUSE cFSs reduce about 50% of the extra user-kernel mode switches compared, that's the main cause of the performance improvement. From the perspective of the storage system, DeFUSE reduces the user-kernel mode switches in the file system layer. Therefore, emerging non-volatile memory with low I/O latency can potentially benefit from this software optimization.

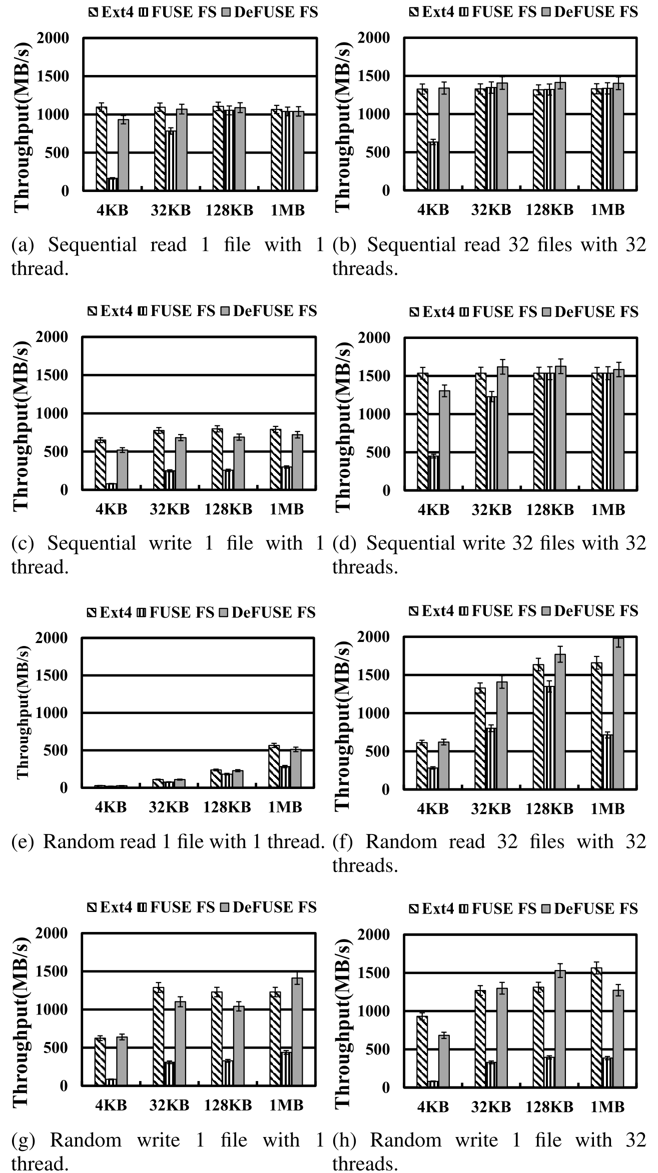
While the DeFUSE framework improves the throughput, the *Data_Map* delivery process in the file metadata requests may cause longer latency. Therefore, we present the latency comparison under the file system configurations, as shown in Figure 7(b). When accessing large files with single thread, the latencies of DeFUSE-based cFSs are less than the FUSE-based cFSs. The reason is that even if the access to one file only brings two *Data_Map* deliveries, the reduced user-kernel mode switches in data requests decrease the latency. In fact, the extra latency brought by the mapping delivery process in the metadata requests can be amortized by multiple data IOs so that the overall file system latency is decreased. We can get the conclusion that DeFUSE-based cFS is suitable for large file access workloads.

C. DATA-CENTRIC WORKLOAD ANALYSIS

Figure 9 shows the throughput result of the data-centric workloads. Below are the detailed observations and analysis.

Sequential Read 1 File With 1 Thread: The throughput of the single thread read workload is shown in Figure 9(a), when the IO size is no less than 32KB, the throughput of *DeFUSE FS* is only 2.4% lower than *Ext4*. When the IO size is 4KB, the throughput of *DeFUSE FS* is 15.0% lower than *Ext4*. We can see that *DeFUSE FS* provides the throughput close to *Ext4*, the main reason is that *DeFUSE FS* benefits from the prefetch read mechanism of FUSE [16]. While the reduced user-kernel mode switches contribute more to the performance improvement when the IO size is less than 32KB.

Sequential Read 32 Files With 32 Threads: When reading 32 files in 32 threads, the throughput is shown in Figure 9(b), we can see that the throughput of *DeFUSE FS* and *FUSE FS* both are larger than *Ext4* when the IO size is no less than 32KB. When the IO size is 128KB, *DeFUSE FS* outperforms



(a) Sequential read 1 file with 1 thread.

(b) Sequential read 32 files with 32 threads.

(c) Sequential write 1 file with 1 thread.

(d) Sequential write 32 files with 32 threads.

(e) Random read 1 file with 1 thread.

(f) Random read 32 files with 32 threads.

(g) Random write 1 file with 1 thread.

(h) Random write 1 file with 32 threads.

FIGURE 9. **The throughput of the data-centric workloads running on *Ext4,FUSE FS* and *DeFUSE FS*. Within each figure, from left to right, the IO size of the workload are 4kB, 32KB, 128KB and 1MB. The results reported here are the average value of the repetitive experiments and the standard deviations of the results here range from 5.2% to 5.8%.**

Ext4 up to 7.6%. The reason for the throughput of *FUSE FS* outperforms *Ext4* is that the underlying storage media is SSD array, so FUSE framework can simultaneously pre-fetch files among different devices, that's why the pre-fetch mechanism works better in multi-threads workload. And the reduced user-kernel mode switches further leads to throughput increment in *DeFUSE FS*.

Sequential Write 1 File With 1 Thread: As shown in Figure 9(c), the throughput of *DeFUSE FS* is 8.9% - 20.0% lower than *Ext4*. Compared to single thread read, the data have to be written to the storage media directly. We can see that *DeFUSE FS* outperforms *FUSE FS* by 2.4X to 6.6X due to the reduced user-kernel mode switches.

Sequential Write 32 Files With 32 Threads: As shown in Figure 9(d), we can see that the throughput of *DeFUSE FS* and *FUSE FS* are larger than *Ext4* when the IO size is no less than 128KB. When the IO size is 128KB, the *DeFUSE FS* outperforms *Ext4* up to 5.8%. The multi-thread writes also benefit from the underlying SSD array since the writes to multiple storage device can be processed in parallel. The reduced user-kernel mode switches further contribute to the throughput improvements of *DeFUSE FS*.

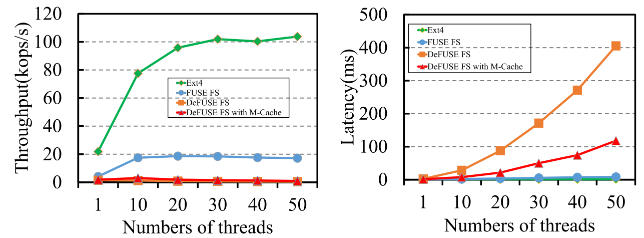
Random Read 1 File With 1 Thread: As shown in Figure 9(e), when random reading files, the read ahead mechanism of FUSE is not working. So when the IO size is 1MB, compared to the throughput of *Ext4*, the throughput of *FUSE FS* causes 49.9% of performance degradation, however the *DeFUSE FS* achieves 90.7% of the *Ext4* throughput. When the IO size drops from 1MB to 4KB, the throughput of these three configurations all decrease from 93.3% to 95.0%. This is because in the random access pattern, with bigger IO size, the read cache can reduce performance degradation.

Random Read 1 File With 32 Threads: When reading one pre-allocated file with 32 threads, *DeFUSE FS* outperforms *Ext4* by 19.3% with 1MB IO size. The performance improvements also benefit from the multiple storage devices and when accessing one file, even in the random access pattern, there are still many read cache hits. However, *FUSE FS* only utilize 43.3% 82.4% of the throughput of *Ext4*.

Random Write 1 File With 1 Thread: Obviously, the underlying SSD array has good performance to the random write workload. Performance degradation caused by *DeFUSE FS* is 14.5% to 26.7%. However, the performance of *DeFUSE FS* outperforms *FUSE FS* by 3.2X to 8.4X. The reduced kernel/user mode switches contribute to the performance improvements of *DeFUSE FS* compared to *FUSE FS*.

Random Write 1 File With 32 Threads: The performance is similar to the workload *rnd-rd-32th-1f*. The same analysis applies here.

Overall Findings: We can see that the write throughput of *DeFUSE FS* and *FUSE FS* reaches peak when the IO size is 128KB, which is exactly the default *big_writes* size in the FUSE framework. The FUSE kernel does not need to split or compact the data when the IO size is 128KB. And when the IO size is 4KB, the throughput of *DeFUSE FS* are far larger than *FUSE FS*, which means that DeFUSE framework performs well even when the IO size is small. When the workload are single thread access of files, the *DeFUSE FS* are little lower than *Ext4*, when the workload are multi-thread access of files, *DeFUSE FS* outperforms *Ext4*.



(a) Throughput comparison on metadata-centric workloads. (b) Latency comparison on metadata-centric workloads.

FIGURE 10. The throughput and latency of the metadata-centric workloads running on *FUSE FS*, *DeFUSE FS* and the *DeFUSE FS* with the M-Cache enabled. The result reported here are the average value of the repetitive experiments and the standard deviations of the results range from 2.3% to 6.8%.

D. METADATA-CENTRIC WORKLOAD ANALYSIS

The throughput and the latency of the metadata-centric workloads are shown in Figure 10. We can see from Figure 10(a), when there is only 1 thread in *File-server* workload, the throughput of *DeFUSE FS* is 35.4% of *FUSE FS*. This is because the small file access causes frequent access to its metadata in user space, thus amplifying the impact of the *Data_Map* delivery process in the metadata request. As the number of threads increases, the throughput of *FUSE FS* increases and reaches a stable throughput at 17,000 ops/s. The throughput of *DeFUSE FS* decreases because multi-thread access brings longer wait time in the DeFUSE metadata request. The operation latency is shown in Figure 10(b). When there is only 1 thread, the latency of *FUSE FS* and *DeFUSE FS* are 0.8ms and 2.6ms respectively. As the number of thread increases, the latency of *FUSE FS* is stable at about 1.8ms, while the latency of *DeFUSE FS* increases up to 405ms. The throughput and latency of *DeFUSE FS* are acceptable in single-thread metadata-centric workload. However, when the number of threads goes up, the latency of *DeFUSE FS* increases up to hundreds millisecond level and the throughput is also greatly decreased.

With the M-Cache enabled in the DeFUSE framework, the throughput of *DeFUSE FS with M-Cache* increases by 1.2X - 2.4X compared to *DeFUSE FS*. The throughput improvement comes from the reduced number of *Data_Map* delivery. Compared to the *FUSE FS*, the throughput increment is not obvious because the M-Cache only caches 1024 of the recently used *Data_Map*, the total amount of the small files are 200,000. The latency of the *DeFUSE FS with M-Cache* is reduced by 47.0% - 75.5% compared to *DeFUSE FS*, the M-Cache significantly reduces the access latency of *DeFUSE FS*. But the latency stills increases up to 118ms when the number of threads increase to 50. We can further increase the maximal count of the M-Cache to reduce the impact on the throughput and latency.

V. RELATED WORK

Performance of the FUSE framework has drawn much attention from the academic research field. Vasily Tarasov et al. discuss that the user-kernel mode switches are the main performance overhead in the FUSE framework [29]. Bharath Kumar Reddy Vangoor et.al further give detailed analysis of the FUSE framework and evaluate FUSE-based file system performance degradation brought by the user-kernel mode switches [16].

Optimization From FUSE Framework Developers: Developers of the FUSE framework have present two optimization method to reduce the kernel/user mode switch overhead: (1) FUSE uses the *splicing* functionality in the Linux kernel [30] to reduce memory copy between the user and kernel space; (2) FUSE provide asynchronous write policy and increase the default write chunk from 4KB to 128KB [17], which reduces kernel/user mode switch in write operation; (3) FUSE uses the file system *prefetch* read mechanism [31] to cache the data to reduce mode switches in read operations. Liu Xin et.al [32] uses the former two optimization methods in their ONFS implementation and achieves performance improvements. DeFUSE can leverage these optimizations to achieve performance improvement.

Optimization From FUSE-Based cFS Developers: Researchers who use FUSE to build prototype file systems have also made optimization to their own customized FUSE-based cFSs. Zhang et.al [11] composite the small files together to improve the file system performance under metadata-centric workload. Shun Ishiguro et al. [33] optimized local file accesses for FUSE-based distributed storage. Yan et al. [12] reduce the kernel/user mode switches in the optical library file system. All of their methods are specific for their own file systems, and can not be extended to general FUSE-based file systems. Besides, their methods are implemented on FUSE version 2.9.X or before, and cannot be applied to the current FUSE framework. Ashish Bijlani et.al [34] present *Extfuse*, an extension framework for FUSE, and Extfuse focuses more on optimizing the performance of metadata operations, and uses in-kernel metadata caching strategy to reduce user-kernel mode switches in the metadata operations.

VI. CONCLUSION

In this paper, we proposed a novel optimization scheme: DeFUSE, which divides the requests in FUSE-based cFS into two types: the metadata and data request, then process these two kinds of requests with different flow. With the carefully designed structure *Data_Map* being delivered to DeFUSE kernel driver in the metadata requests, the file data requests can be completed within the kernel so that the unnecessary user-kernel mode switches in the data requests are removed, thus greatly improving the throughput. We implement the DeFUSE framework and port three FUSE-based cFSs to DeFUSE-based cFSs with minor modification of their code. The experiment result shows that for data-centric workloads, the bandwidth of DeFUSE-based cFSs outperform the corresponding FUSE-based implementations by 1.8X to 3.8X.

REFERENCES

[1] (2018). *Ext4 Wiki*. [Online]. Available: https://ext4.wiki.kernel.org/index.php

[2] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *Proc. FAST*, 2015, pp. 273–286.

[3] Z. Wiki. (2017). *Zf*. [Online]. Available: https://wiki.ubuntu.com/ZFS

[4] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using Crash Hoare logic for certifying the FSCQ file system," in *Proc. 25th Symp. Operating Syst. Princ.*, Oct. 2015, pp. 18–37.

[5] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "All file systems are not created equal: On the complexity of crafting crash-consistent applications," in *Proc. 11th Symp. Operating Syst. Design Implement. (OSDI)*, 2014, pp. 433–448.

[6] M. Szeredi. (2010). *Fuse: Filesystem in Userspace*. [Online]. Available: https://lwn.net/Articles/118574/

[7] (2018). *Ceph Client*. [Online]. Available: https://github.com/ceph/ceph

[8] (2014). *Mounting HDFS With Fuse*. [Online]. Available: https://wiki.apache.org/hadoop/MountableHDFS

[9] (2018). *A User-Space File System for Interacting With Google Cloud Storage*. [Online]. Available: https://github.com/GoogleCloudPlatform/gcsfuse/

[10] P. Braam and D. Bonnie, "Campaign storage," in *Proc. Int. Conf. Massive Storage Syst. Technol.*, 2017, pp. 1–8.

[11] S. Zhang, H. Catanese, and A.-I. A. Wang, "The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance," in *Proc. FAST*, 2016, pp. 15–22.

[12] W. Yan, J. Yao, Q. Cao, C. Xie, and H. Jiang, "ROS: A rack-based optical storage system with inline accessibility for long-term data preservation," *ACM Trans. Storage*, vol. 14, no. 3, Nov. 2018, Art. no. 28.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass storage Syst. Technol. (MSST)*, May 2010, pp. 1–10.

[14] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th Symp. Operating Syst. Design Implement. (OSDI)*, 2016, pp. 265–283.

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, nos. 10–10, p. 95, Jun. 2010.

[16] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To fuse or not to fuse: Performance of user-space file systems," in *Proc. FAST*, 2017, pp. 59–72.

[17] (2016). *Fuse Big_Writes*. [Online]. Available: https://github.com/libfuse/libfuse/releases

[18] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proc. Conf. High Perform. Comput. Netw., Storage Anal.*, New York, NY, USA, Nov. 2009, Art. no. 21. doi: 10.1145/1654059.1654081.

[19] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li, "Locofs: A loosely-coupled metadata service for distributed file systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2017, Art. no. 4.

[20] Y. Chen, C. Li, M. Lv, X. Shao, Y. Li, and Y. Xu, "Explicit data correlations-directed metadata prefetching method in distributed file systems," *IEEE Trans. Parallel Distrib. Syst.*, to be published.

[21] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira, "SafeFS: A modular architecture for secure user-space file systems: One FUSE to rule them all," in *Proc. 10th ACM Int. Syst. Storage Conf.*, May 2017, Art. no. 9.

[22] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Design Implement.*, Nov. 2006, pp. 307–320.

[23] M. E. Hoskins, "SSHFS: Super easy file access over SSH," *Linux J.*, vol. 2006, no. 146, p. 4, Jun. 2006.

[24] (2018). *NTFS-3G*. [Online]. Available: https://github.com/osxfuse/osxfuse

[25] (2018). *S3fs-Fuse*. [Online]. Available: https://github.com/s3fs-fuse/s3fs-fuse

[26] V. T. G. Amvrosiadis. (2018). *Filebench*. [Online]. Available: https://github.com/filebench/filebench/wiki

[27] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *Login, USENIX Mag.*, vol. 41, no. 1, pp. 6–12, Mar. 2016.

[28] (2011). *SSD Quality and Performance Comparison [EB/OL]*. [Online]. Available: http://www.technical-direct.com/en/ssd-quality-and-performance-comparis% on-a-testing-and-evaluation-report-for-ssds-in-the-market-today-2-2/

[29] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra incognita: On the practicality of user-space file systems," in *Proc. HotStorage*, 2015, pp. 1–5.

[30] L. Torvalds. (2017). *Linux splice*. [Online]. Available: http://man7.org/linux/man-pages/man2/splice.2.html

[31] R. A. J. Griffioen, "Reducing file system latency using a predictive approach," in *Proc. USENIX Conf. File Storage Technol.*, Jun. 1994, pp. 197–207.

[32] X. Liu, Y.-T. Lu, J. Yu, P.-F. Wang, J.-T. Wu, and Y. Lu, "ONFS: A hierarchical hybrid file system based on memory, SSD, and HDD for high performance computers," *Frontiers Inf. Technol. Electron. Eng.*, vol. 18, no. 12, pp. 1940–1971, Dec. 2017. doi: 10.1631/FITEE.1700626.

[33] S. Ishiguro, J. Murakami, Y. Oyama, and O. Tatebe, "Optimizing local file accesses for FUSE-based distributed storage," in *Proc. SC Companion, High-Perform. Comput., Netw., Storage Anal. (SCC)*, Nov. 2012, pp. 760–765.

[34] A. Bijlani and U. Ramachandran, "Extension framework for file systems in user space," in *Proc. Annu. Tech. Conf. (ATC)*, Renton, WA, USA, 2019, pp. 121–134. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/bijlani

WENRUI YAN received the B.S. degree in computer science major from the Huazhong University of Science and Technology (HUST), China, in 2013. He is currently pursuing the Ph.D. degree in computer architecture with the Wuhan National Laboratory for Optoelectronics (WNLO). His research interests include optical library storage systems, and file system design and implementation.

JIE YAO received the B.S. degree in computer science major and the M.S. and Ph.D. degrees in computer architecture from the Huazhong University of Science and Technology (HUST), China, in 2001, 2004, and 2009, respectively, where he has been a Lecturer of computer architecture, since 2010. His research interest includes the methodology and implementation of computer architecture, especially on the long-term data preservation systems.

QIANG CAO received the B.S. degree in applied physics from Nanjing University, in 1997, and the M.S. degree in computer technology and the Ph.D. degree in computer architecture from the Huazhong University of Science and Technology, in 2000 and 2003, respectively. He is currently a Full Professor with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. His research interests include computer architecture, large scale storage systems, and performance evaluation. He is a Senior Member of the China Computer Federation (CCF), IEEE, and ACM.