User-level Parallel File System: Case Studies and Performance Optimizations

Journal:	*Concurrency and Computation: Practice and Experience*
Manuscript ID	CPE-21-0475.R3
Editor Selection:	Regular Issue Submission
Wiley - Manuscript type:	Research Article
Date Submitted by the Author:	21-Dec-2021

Complete List of Authors:	Zou, Yanliang; ShanghaiTech University, School of Information Science and Technology; Shanghai Institute of Microsystem and Information Technology; University of the Chinese Academy of Sciences Chen, Chen; ShanghaiTech University Deng, Tongliang; ShanghaiTech University Zhang, Jian; ShanghaiTech University Xiaomin, Zhu; National University of Defense Technology Chen, Si; West Chester University of Pennsylvania Yin, Shu; ShanghaiTech University

Keywords:	User-level File Systems, Parallel File System, FUSE, I/O performance, Storage System, Data Consistency

Note: The following files were submitted by the author for peer review, but cannot be converted to PDF. You must view these files (e.g. movies) online.

User-level Parallel File System_ Case Studies and PerformanceOptimizations .zip

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

User-level Parallel File System: Case Studies and Performance Optimizations

Yanliang Zou[1,2,3] | Chen Chen[1] | Tongliang Deng[1] | Jian Zhang[1] | Xiaomin Zhu[4] | Si Chen[5] | Shu Yin*[1]

[1]ShanghaiTech University, Shanghai, China
[2]Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, Shanghai, China
[3]University of Chinese Academy of Sciences, Shanghai, China
[4]State Key Laboratory of High Performance Computing, Changsha, China
[5]West Chester University of Pennsylvania(West Chester), PA, United States

Correspondence
*Shu Yin. Email: yinshu@shanghaitech.edu.cn

Present Address
393 Middle Huaxia Road, Pudong, Shanghai, China, 201210

Summary

User-level file systems are usually adopted to bridge the gap between efficacy and efficiency of file system developments for new applications' I/O demands. And the widely known user-space file system framework, FUSE, is commonly utilized to deployed user-level file systems. This paper first uses a popular stack-able file system as a case study to exam how FUSE affects I/O performance. Based on the testing and analytical results, this paper then presents SHC, an implementation method to implement a user-level file system without FUSE intervention. Experimental results indicate that SHC improves write bandwidth by up to 5.6x compared with that of FUSE and present leading superiority on read cases.

KEYWORDS:
User-level File Systems, Parallel File System, FUSE, I/O performance, Storage System, Data Consistency

1 | INTRODUCTION

Parallel and distributed file systems provide a single-node massive storage abstraction and distribute files in a striped manner. Existing parallel file system such as Lustre[1], GPFS[2] and OrangeFS[3], are widely used in high performance computing (HPC) and other concurrent scenarios. Besides, some lightweight parallel file systems are designed and developed to serve specific functions, for example, Parallel Log-structured File System (a.k.a. PLFS)[4] can improve the I/O performance under the N-1 access pattern. These lightweight file systems are implemented in user space laying on an existing file system to offer data service, which can greatly reduce the difficulty of development.

Unlike those implemented in kernel-space, these user-level file systems can not benefit from the unified POSIX interface which is the most widely used I/O interface in applications. Thus, they always seek help from some file system frameworks like FUSE (a.k.a. Filesystems in User-space)[5]. FUSE is a popular user-level file system framework that allows users to deploy file systems without modifying kernels. It receives requests from applications via the FUSE kernel module and forwards them to a corresponding user-space handler. Therefore, every I/O operation leads to several kernel crossings. The performance with FUSE would be affected considering the further delays from queuing time and request reorganizing time in the FUSE layer[6]. Vangoor et al. studied performance characteristics of an ext4 file system on FUSE and demonstrated that FUSE results in inconsistent workload-dependent slowdowns[7].

We conducted a preliminary *fs_test*[8] benchmark test on a 4-node Lustre cluster with and without PLFS, a user-level file system that is designed to accelerate N–1 writes by transforming random, dispersed N–1 writes into sequential N–N writes in a log-structured manner. The Lustre cluster is equipped with 1 MDS and 4 OSSes that are interconnected by Mellanox®Inifiband switch MT27500 ConnectX-3 (56 Gpbs). Each OSS server has two Intel®Xeon®CPUs E5-2650 v2 @2.6GHz and 64 GB

DRAM. The total capacity of the cluster is 71 TB. We use *fs_test* to create 16 processes and write various sizes of data into a single file concurrently. We found that PLFS-assisted Lustre provides a much better I/O bandwidth (up to 3x) when the data size is larger than 64MB (see Fig. 1). Fig. 1 also reveals the limits of PLFS in handling smaller-sized data mainly due to the overhead of multiple kernel crossings by the FUSE layer. We study the impact of FUSE in detail in Section 3.

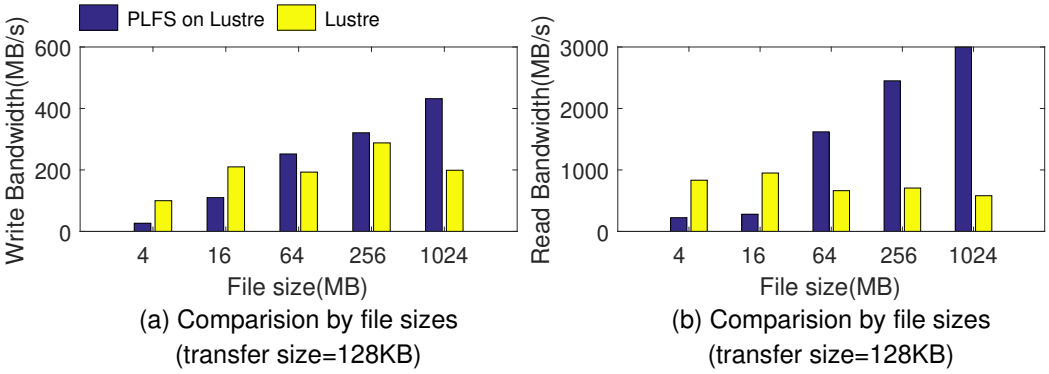

FIGURE 1 A performance comparison with and without a user-level file system

In addition, following the recent discovery and patching of Meltdown[9], a currently Intel-specific security issue, kernel programmers were forced to isolate kernel and user memory space more thoroughly, exacerbating kernel crossing penalties due to page walking and TLB flushes. Given that existing large-scale computing systems can not replace flawed processors immediately, it is inevitable that workloads highly dependent on system calls will suffer slowdowns. It would be highly unsatisfactory if the performance overhead resulting from utilizing FUSE takes a significant proportion of the total I/O processing time within applications running on large-scale computing systems.

Data consistency has to be maintained as the underlying file system goes parallel or distributed. MPI-IO technique can handle the data consistency issue well, but it requires additional efforts to change APIs from standard POSIX. Switching APIs to MPI-IO may become a challenge for domain scientists who do not have rich experience in API code modifications. They are inclined to standard POSIX API to focus on the functionalities of their codes. Besides, domain scientists may not take advantage of MPI-IO as it requires execution parameter tuning for different computing system setups. Without technique support from experienced I/O teams, it would be hard for domain scientists to benefit from MPI-IO. It motivates us to design a mechanism that could be serving as an expedient that takes advantage of MPI-IO while retaining the POSIX interface. Users like domain scientists then do not bother to spend efforts on code modifications to switch to MPI-IO while taking benefits of parallelism.

In this paper, we first conduct a case study [10] to examine the effects of FUSE on the performance of user-level file systems. Then we implement a preliminary solution for "PLFS without FUSE" using a dynamic library. We further notice that dynamic library approaches can not guarantee data consistency especially when newly generated data is appended to a file concurrently. Appending data is a common I/O operation in modern parallel and distributed storage systems that can be maintained with advanced interfaces like MPI. Recall that our dynamic library approach intercepts and redirects requests from users to the underlying file systems by overriding general system call under POSIX. If we can separate the processing of metadata and data, consistency can be ensured without introducing additional overheads. We update an approach and name it SHC [11]– Synchronization Processing Server (SPS), Hooking Library (HLib) and Customed IOStore (CStore).

SPS is to deal with the write synchronization issue when multiple devices are writing to a single file concurrently, and it retains data consistency without introducing additional memory copy and data buffers. SPS also maintains the file handles and offsets so that every device knows where to write its data in the file.

HLib is a library that captures I/O requests, distills metadata operations, and dispatches them to SPS to locate data offsets. Upon the returns from SPS, HLib forwards the data offsets to CStore to proceed with data writing and reading.

CStore is a modified interconnect module that sits between user-level file systems and underlying file systems. CStore delivers data read and write operations to the underlying file system through interfaces like POSIX or parallel file system APIs. CStore is linked to **SPS** and coordinates data operations with HLib.

The proposed SHC aims at 1) reducing FUSE intervenes for user-level parallel file systems while avoiding additional code modifications to applications, 2) maintaining the data consistency for scenarios where multiple processes write to a single file without communicating with each other, and 3) providing parallel I/O accesses in user-level file systems under POSIX standard. The major contribution of this paper is three-folded:

- We use a stackable user-level parallel file system (a.k.a (*sPFS*)) named PLFS as a case study to examine how FUSE affects I/O performance;
- We propose SHC, an implementation method for sPFS that reduces FUSE overhead. SHC is designed to avoid FUSE intervention and maintain data consistency for scenarios where multiple processes write to a single file asynchronously;
- We implement an SHC prototype and integrate it with PLFS into a real-world cluster;
- A comprehensive experimental study is provided to evaluate the efficacy of SHC.

The rest of the paper is organized as follows: Section 2 addresses our motivation and background information of FUSE and PLFS. Section 3, 4 describes the design and principles of our proof of concept FUSE-bypass mechanisms for PLFS. Section 5 presents the experiment setup and results, shows the analysis of technical factors which affect the performance in detail. Section 6 discusses the potential use cases and possible drawbacks of FUSE. Section 7 presents an overview of related works. Section 8 concludes the paper with a summary of the broad issues.

2 | BACKGROUND AND MOTIVATION

2.1 | Background

User-level File System Framework

In modern operating systems, most file systems reside in the operating system kernel as its operations require raw access to storage or network devices using a higher privilege level. However, the user may wish to place an interposition layer for virtual file manipulation. These file systems do not store data; they manipulate files and reorganize them for the underlying storage systems, and we call them stackable file systems. These virtual file systems provide a simple file abstraction for many applications and are fairly easy to implement without having to write kernel modules or drivers.

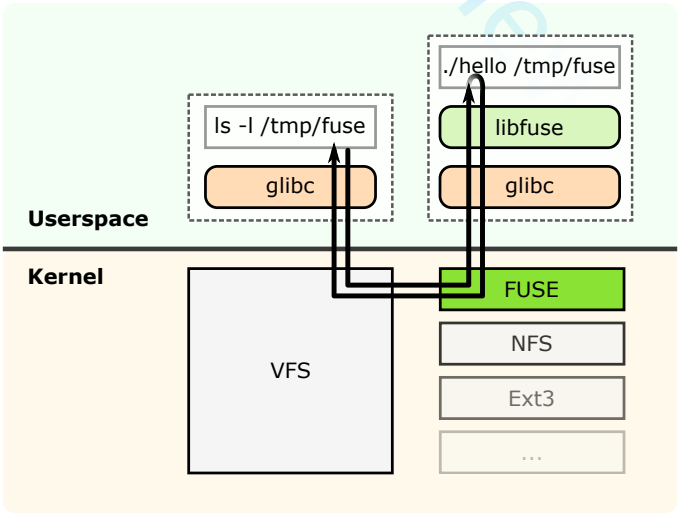

FIGURE 2 FUSE operation routine

Filesystem in User-space (FUSE) is a common interface found on many UNIX systems that allow users to implement file systems running in user space without modifying the kernel. FUSE bridges the user space file system and user applications for

normal I/O operations. Fig. 2 [12] shows the operation principles of FUSE. FUSE consists of two parts, the kernel module and user-level daemon (the instance of a FUSE file system). They communicate via a virtual device `/dev/fuse`. When an I/O operation is invoked on a FUSE-based virtual file system, the request passes through the kernel virtual file system (VFS). The kernel then passes the request to the FUSE kernel module, which processes the request and redirects it to a user-level FUSE daemon. The daemon performs operations on the underlying file systems and redirects the result back to the user application via the kernel, which may trigger system calls again.

While the file system is running in user space, developers and users do not need to struggle with the kernel since FUSE will handle all this stuff. User-space file systems like sshfs [13], GlusterFS [14] and CRFS [15] that operate over FUSE highly extend the use case of file systems. This is a major contribution to FUSE. But this method also introduces more kernel crossings, more memory copies, and a higher possibility that the I/O request would be serialized.

Attempts have been made to optimize FUSE processing speeds [16], but studies show that the performance of FUSE-mounted native file systems is highly inconsistent even with optimization [7], ranging from marginally higher than native file systems to vast slowdowns. These performance characteristics indicate FUSE does have its pros and cons.

Parallel Log-structured File System

Massively parallel applications running across thousands of processors requires robust error protection mechanisms and failure recovery systems due to the high failure rates that supercomputers and data centers experience. Checkpointing is one such technique that periodically saves data to persistent storage, so that, when failures occur and the system is reset, the application can continue from the last checkpoint. For most applications, writing into a single file is the most convenient method of storing a program state. However, these writings tend to be small and unaligned with file system boundaries, resulting in poor performance.

The Parallel Log-structured File System (PLFS) [4] attempts to solve this problem by inserting an interposition layer in the storage stack to rearrange storage patterns from the user application to patterns to achieve better performance with parallel storage devices. The original N-1 write pattern can be reorganized into an N-N pattern, improving checkpoint bandwidths significantly.

2.2 | Motivation

FUSE is to assist applications to utilize user-level file systems without introducing new customized APIs. Any customized API other than standard POSIX would introduce additional efforts on code modification, which are very unfriendly to application developers. It is also infeasible to modify scientific applications' code to work with a new API. On the operating system level, I/Os are mostly completed by POSIX APIs since POSIX is a standard interface. Even though in the HPC cluster, POSIX is used as the unified interfaces of the parallel and distributed file systems and MPI-IO lays between HPC applications and POSIX interfaces as a middleware. MPI-IO is a well-known parallel programming model to assist scientific applications for better parallel I/O performance, but it requires applications to manage cache coherency themselves. That's why many HPC machines use MPI or POSIX with MPI-IO for data consistency. For example, the TaihuLight Supercomputer machine stacks MPI-IO on top of a distributed file system called LWFS [17] via the POSIX interface.

Besides, many HPC machines employ customized MPI libraries for their dedicated architectures. It is getting harder for developers to modify and recompile a customized MPI library for their applications. For example, the TaihuLight supercomputer uses a tailored MPI version called SunwayMPI, which contains tuned functions for the Sunway architecture and may not be fully compatible with the open-sourced MPI. Without proper modification of the MPI library, the patches prepared for the open-sourced MPI library may not work correctly on a customized MPI. Without an experienced code optimization team, using POSIX API could be an efficacy approach for regular users.

File systems implemented in kernel-space can enjoy the convenience of POSIX which is supported by Linux kernel natively. User-level file systems, however, commonly depend on frameworks like FUSE to offer POSIX services and register themselves on VFS. Vangoor et al. did explore the performance characteristics of a local ext4 file system on FUSE in their research [7]. Nevertheless, more and more CPU vulnerabilities have been found by computer security researchers. For example, Meltdown is a well-known CPU bug discovered by several security researchers [9] affecting Intel CPUs that implement speculative and out-of-order execution models, basically meaning from Intel Pentium Pro processors to current models. To fix these vulnerabilities without replacing the affected hardware, we can only use a software-level patch. This method may cause performance issues. The current solution for Meltdown on Linux systems is the Kernel Page Table Isolation (KPTI) mechanism, which unmaps kernel memory mappings from the user application page table, leaving minimal kernel data exposed in attacks. We have tested the performance loss of KPTI over PLFS via FUSE on a local ext4 file system using *ior* [18], a file system I/O benchmark tool. As

is indicated in Fig. 3, the preliminary results show that these patches introduce significant slowdowns for PLFS with FUSE. The performance loss in the worst case could reach 20% in our tests. This result indicates that FUSE suffers from higher kernel/user memory isolation on the operating system level. Though this is a simple experiment, we believe that the more strict privilege level protection may call for more consideration on the performance issue of FUSE, especially on heavy parallel I/O workloads.

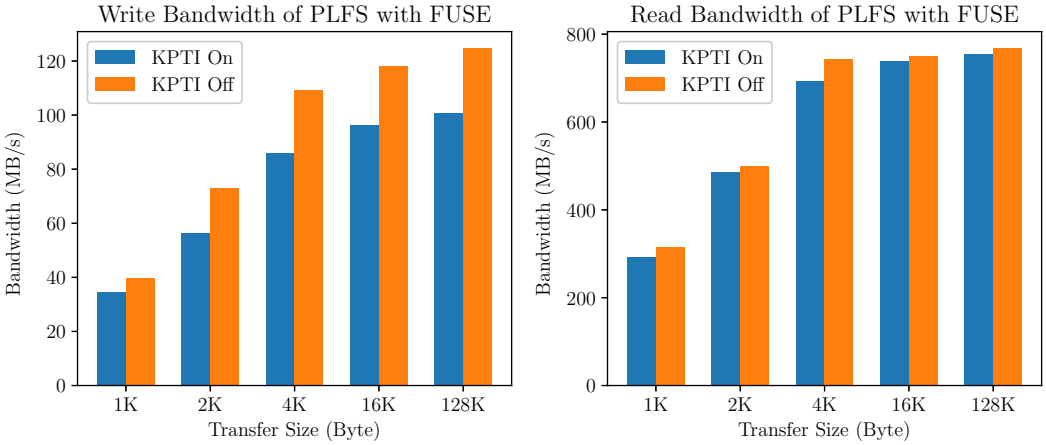

FIGURE 3 Performance of PLFS via FUSE over KPTI status

3 | LIBPLFS: A CASE STUDY OF FUSE

To discover how FUSE affects the performance of a storage system, we choose PLFS as our case study because it is a popular efficient parallel file system. PLFS is optimized for parallel I/Os, so we could use PLFS to evaluate the performance of FUSE under highly parallel workloads regardless of the parallel bottleneck introduced by the actual file system behind FUSE. What's more, PLFS does not have an official kernel module that could make it able to be mounted as a kernel file system, which means we can implement a solution for "PLFS without FUSE". The development process enriched our knowledge on FUSE and those underlying I/O operations of an operating system, which helps our further discussion on the performance issue of FUSE under parallel workloads.

Wright *et al.* previously implemented a library called LDPLFS[19], which could be used to improve the I/O performance of PLFS without modifications to the application. It uses dynamic linking to inject system calls and then reorganize the I/O routines. LDPLFS does not offer a mechanism to handle consistency problems which is unnecessary in HPC centers because of tools like MPI-IO. It depends on the current application process and does not serve communication between other processes. In other words, PLFS with LDPLFS is strongly coupled with MPI-IO and it can not independently serve as a file system, which greatly limits the deployment of PLFS and other user-level file systems with a similar dynamic library.

But inspired by the idea of dynamic linking I/O redirect mechanism, we implemented a pre-loaded dynamic library, *libplfs*, which is the predecessor of HLib(see Section 4.1.2). With this library that exempts FUSE from PLFS, we could evaluate the performance overhead of FUSE. And with our own diagnostic code within the library, we could analyze the performance details of PLFS.

3.1 | Main Scheme

The fundamental idea of our implementation is to use a dynamic library to redirect the I/O routines of PLFS, then take critical jobs that could be done in user-space back to the user-space. The avoided overhead of kernel crossings and VFS I/O reorganization could introduce more capability to highly concurrent I/Os. Our key ideas could be summarized into two parts.

Dynamic linking: Dynamic linking allows the program to link to an external library at runtime, allowing us to redirect the I/O routines of a user application efficiently. The `LD_PRELOAD` environment variable defines the very first loaded dynamic library. A user-customized dynamic library could have the highest priority in dynamic linking, even higher than standard libraries (like *glibc* in Linux), such that all the I/O function calls could be redirected to a user-customized function. This method gives user programs a great convenience to reorganize their I/O routine.

User Space API Utilization: Kernel crossings introduce more overhead, so we want to migrate things back to the user-space. For example, a big write request will first come into the memory space of FUSE, then be reorganized by FUSE and sent back to the user space PLFS daemon. The extra memory copies are excessive and not necessary. What's more, the file status of a process is usually maintained by the operating system kernel when the file system that these files are present is in the kernel space. In PLFS, the file status is maintained by the FUSE kernel module if it is mounted via FUSE. We want to take things back to user-space, eliminating those operations which are unnecessary to be done in the kernel. File systems implemented via FUSE usually have a set of user-space APIs since their user-space daemon will utilize its functionality in user-space. Utilizing user-space APIs of such a file system could take its job back to the user-space, which may eliminate the overhead introduced by FUSE.

3.2 | Methodology

PLFS provides a series of user-level APIs that allow direct access to PLFS functions. Without the assistance of the FUSE layer, developers have to port POSIX I/O APIs to PLFS-specific ones and recompile the code before the execution. This approach requires expertise and labor in code modification and usually is not viable for high-performance platforms. Thus, we implement a file mapping table in user space memory to maintain the file status.

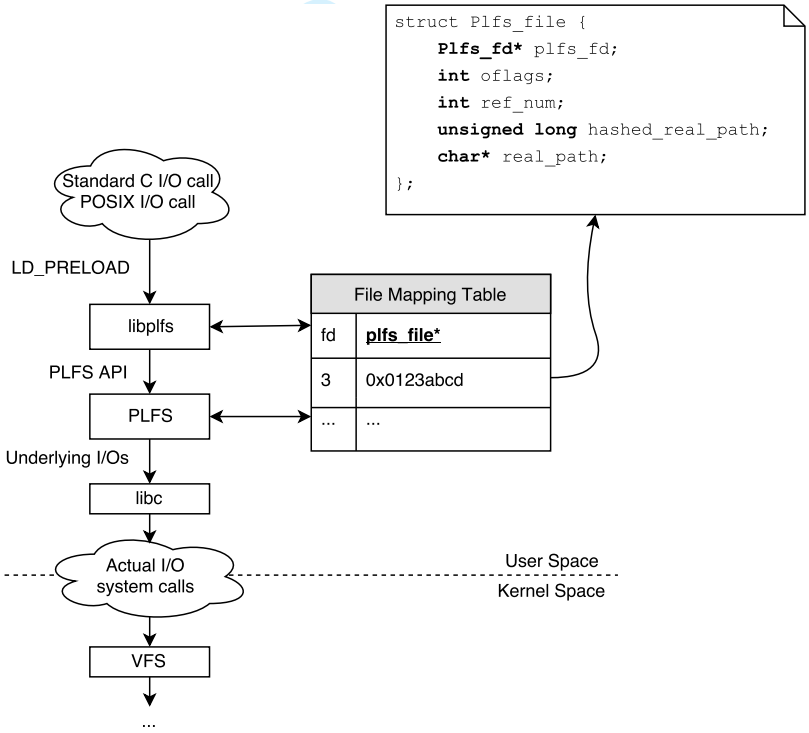

FIGURE 4 Process Flow of *libplfs*

We build the library from scratch with the help of the PLFS API. The basic idea of *libplfs* is to translate a standard C or POSIX I/O function call to a PLFS API function call and maintain the file status table in the application's memory space. Fig. 4 visualized the general I/O routine of *libplfs*.

Take `open()` as an example, *libplfs* opens a temporary virtual file on the local file system in advance, then a corresponding file descriptor will be allocated (e.g. in Fig. 4, a file is opened with *fd*=3 at the end). *libplfs* then calls `plfs_open()` to get a

PLFS file descriptor (e.g. *plfs_file**=0x0123abcd in Fig. 4) and save the mapping relationship between the virtual file (*fd*) and the PLFS file (*plfs_file**). Since PLFS is transparent to the applications, applications can only be aware of the virtual file (*fd*) while the actual data will be stored in the PLFS file. When an application tries to read or write to a file (*fd*), according to the mapping, our library will instead invoke the read or write function for the PLFS file(*plfs_file**), then set the virtual file's (*fd*) offset accordingly. When closing a file, we close the virtual file descriptor (*fd*) and free all related memory, including the file descriptor mapping table entry and, if necessary, the C `FILE` structure. For concurrent I/Os, for example, opening a file multiple times by different threads, our PLFS file structure will record the reference number and utilize PLFS's design on multiple operations on the same file. This won't cause too much memory usage.

Before every I/O operation, our library will sanitize the file-specific parameter that is transferred in, as we need to differentiate between the normal I/O calls and calls to PLFS. If we determine it is a PLFS operation, the library will execute the procedure for PLFS. Otherwise it will call `dlsym()` to determine the address of the true `open()` function in *libc*, jump to that and process the non-PLFS operation.

Libplfs overrides the most commonly used standard C library I/O functions, allowing it to handle common situations. The user would simply have to specify the `LD_PRELOAD` environment variable when executing the desired program, pointing it to the pre-compiled *libplfs* dynamic library. Afterward, PLFS could be utilized directly by calling POSIX I/O functions or standard library I/O functions, exempting FUSE. This methodology can be extended to most of the executable user space applications which dynamically link *libc* at runtime. This dynamic library could even be deployed system-wide because it can handle both PLFS and non-PLFS I/Os.

In the perspective of design, *libplfs* could avoid excessive kernel crossings and kernel function calls introduced by the FUSE I/O mechanism. However, there are potential drawbacks of *libplfs*. A huge difference between FUSE and *libplfs* is the privilege level. FUSE resides in the kernel, so in kernel's view, it is a kernel file system, which means it can utilize kernel page cache while *libplfs* in user-space cannot.

3.3 | Performance Evaluation of FUSE

We set up and configure our experimental cluster for performance evaluations. The cluster consists of five nodes - one master node and four HDD storage nodes. The node specification table is shown in Table 1.

TABLE 1 Cluster node specification

Node	Master Node (1x)	Storage Node (4x)
CPU	Intel(R) Xeon(R) E5-2603 v4 @ 1.70GHz	Intel(R) Xeon(R) E5-2603 v4 @ 1.70GHz
Memory	16GB DDR4 2400MHz	16GB DDR4 2400MHz
Storage	2 * 1TB HDD, 1 for storing config files	4 * 1TB HDD, 3 for Ceph storage pool
OS	Linux Mint Sonya with 4.8.0-53-generic kernel	Linux Mint Sonya with 4.8.0-53-generic kernel

PLFS is a stackable parallel file system, as it does not store files, but instead rearranges files for the underlying file systems. We chose CephFS[20] to be PLFS's back-end file system due to its reliability and efficiency on massive concurrent I/Os. The four storage nodes constitute a Ceph object storage cluster. With one node being chosen to be the meta-data server, the storage cluster serves as a remote file system. We mount this file system via the Ceph kernel module on the master node and specify the mounting point as PLFS's back-end file system.

We first mount PLFS(v2.5) via FUSE with default settings and test the performance as a control group. When testing the performance of *libplfs*, we specify the `LD_PRELOAD` environment variable to the compiled *libplfs* dynamic library before executing the benchmark via `mpirun`. With the help of *libplfs*, all the I/O routines are redirected without being intervened by the FUSE, and the results of the testing tool are the performance information of PLFS with *libplfs*. Lastly, we mount PLFS via FUSE with kernel page cache disabled, to see how kernel page cache would affect FUSE's performance.

So there are three different experiment conditions: FUSE with cache, FUSE with direct I/O (no cache), and no-FUSE (FUSE bypassed via *libplfs*). The backend Ceph file system is always mounted via the Ceph kernel module with cache enabled, this is for a better simulation on real-world systems, as they often utilize cache mechanism greatly.

The benchmark tool we are using in the paper is *fs_test*[8], which is an open-source I/O pattern emulation benchmark application developed at LANL. This benchmark can be used to emulate a real application I/O pattern. It supports the MPI/IO, POSIX, and PLFS API I/O interfaces. The concurrent I/O follows the N-1 model. The type of I/O workload won't be a potential bottleneck since PLFS is designed to transfer the N-1 workload to the N-N workload. Every read experiment is right after the corresponding write experiment to make the backend file system warm-up before doing a reading test. This is to better utilize backend cache, which avoids a potential backend file system bottleneck. After each writing test, the file system will do a synchronization, to flush the memory, and for parallel file systems usually, build the indices.

In the aforementioned sections, we addressed a difference between FUSE and *libplfs* that the latter does not equip with a kernel-level cache while the former makes use of the cache to take advantage of prefetching on reads or buffering on writes. *libplfs* does not implement the cache mechanism because of the limited access permissions in the user-space. Although the I/O performance may suffer from the lack of caches, *libplfs* is beneficial by reducing the number of kernel crossings and memory copies, which are major overheads to I/O accesses. Our experiment results show how the factors affect the performance of parallel file systems in userspace.

We choose some representative results which are categorized as follows:

- Small file I/O (128MB) and big file I/O (2GB)
- Serial I/O (1 thread) and concurrent I/O (4 threads)
- Read and write test provided by the benchmark tool
- Small transfer sizes (1KB, 4KB, 16KB) and relatively bigger transfer sizes (64KB, 256KB, 1MB)

The argument we choose on the x-axis is the transfer size because: 1) the transfer size has a strong relationship to cache utilization, 2) it will also determine the system call counts and kernel function call counts under the same file size. We ran every single test five times and use error bars to indicate the errors and variance of the results.

Fig. 5 shows the read benchmark results. The read performance of no-FUSE outperforms FUSE with direct I/O regardless of transfer size. Furthermore, FUSE with cache performs better than no-FUSE and FUSE with direct I/O in tests with small transfer sizes (smaller than 32KB). When the transfer size gets bigger (larger than 32KB), *libplfs* outperforms FUSE with cache while FUSE with cache seems to reach its plateau, but FUSE with cache is always better than FUSE with direct I/O on the read performance tests. In addition, FUSE with direct I/O and no-FUSE have the same trend with the changing transfer size while FUSE with cache performs more stable. But the results of single thread read and multiple threads read are similar.

Fig. 6 shows the write benchmark results. The write performance of FUSE direct I/O outperforms FUSE with cache regardless of transfer size. And the performance of no-FUSE is generally better than the two FUSE-based I/O methods when the transfer size is small (smaller than 64KB). When the transfer size is getting bigger, no-FUSE is outperformed by FUSE direct I/O, but it is still better than FUSE with the cache. When the transfer size is small, the performance gap between FUSE and no-FUSE is more obvious in the multi-threaded test than in the single-threaded test. That is, no-FUSE gains better performance when concurrent small writes occur.

Some results shown in the previous part are expected. We expected that no-FUSE will outperform FUSE all the way except reading with a small transfer size due to the page cache utilization, but some results are not as expected. A potential problem here is "double caching". The kernel recognizes FUSE and the underlying file system as two different file systems, so they have separate cache space in the kernel. When CephFS in our test has a clean buffer, doing the reading test will cause the file content to be cached for two copies. Secondly, the cost of system calls should be considered. Especially on the condition that the actual I/O time cost is not high, the overhead introduced by privilege level switching is more apparent.

4 | SHC: A FUSE PERFORMANCE OPTIMIZATION APPROACH

However, the dynamic link approach(*libplfs*), is insufficient to be a solution to bypass FUSE for a user-level parallel file system. Concurrency on a file can not be handled since *libplfs* is attached to different independent processes each of which maintains

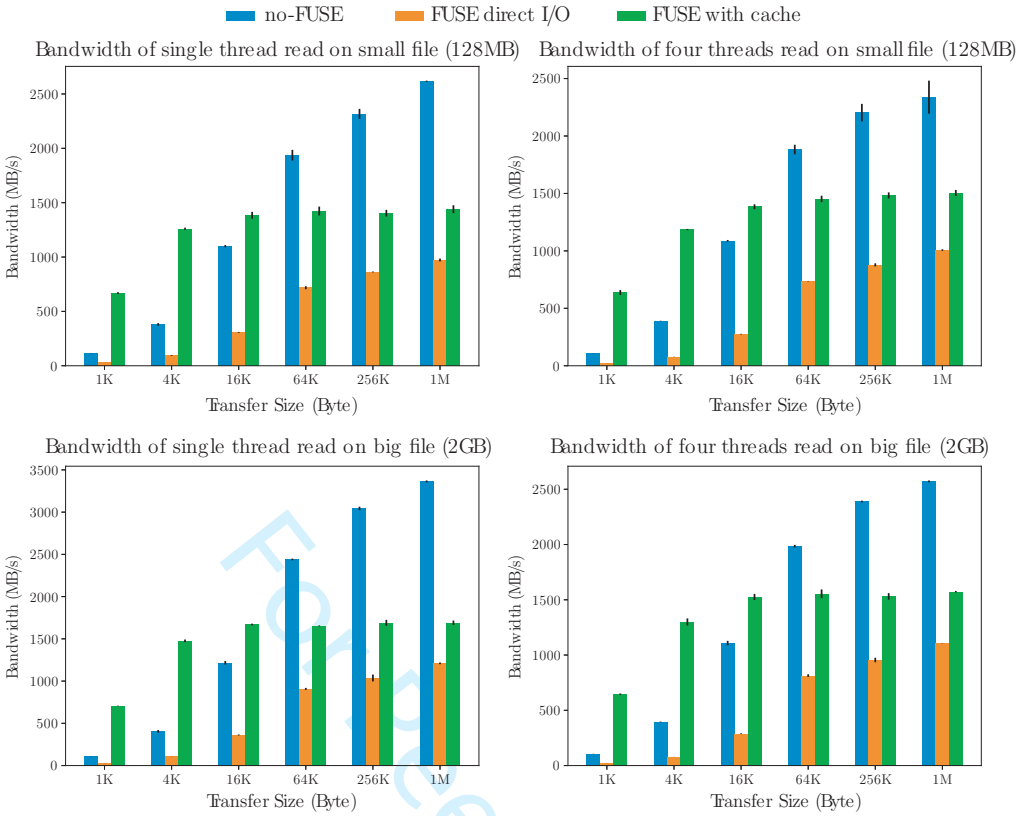

FIGURE 5 Read performance comparison

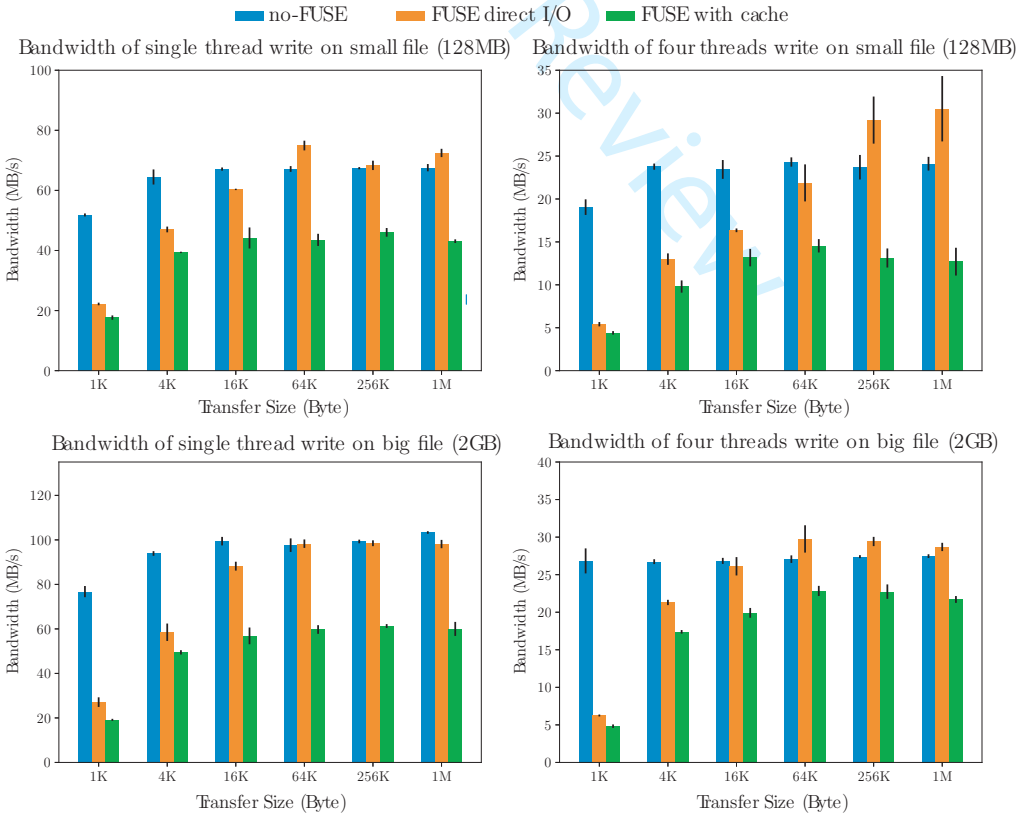

FIGURE 6 Write performance comparison

a copy of file metadata such as offset and file size in its individual memory. In general, a dynamic library is a self-supporting library and it can not be a complete method in user-space because no communication among the working processes to ensure consistency. This problem comes acute when multiple processes simultaneously append data to the same file. Appending data is a common writing behavior in the POSIX environment by setting an `O_APPEND` flag when opening a file. With this flag, before each writing operation, the kernel will adjust the file offset to the end for the coming write. Therefore, the user-level file systems need other mechanisms to arrange this field in user-space. A self-supporting dynamic library can not handle this case since the application processes can not communicate with each other for instant file size.

In this section, we extend *libplfs* to an advanced approach—-the proposed SHC idea. SHC mainly aims at mounting user-level parallel file systems without FUSE intervention. In addition to the FUSE detour, SHC provides a write synchronization alternative for scenarios where independent users are trying to write to the same file.

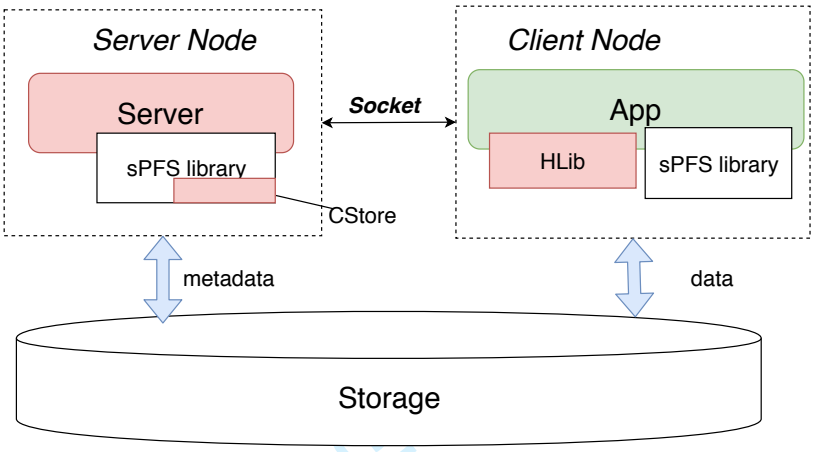

FIGURE 7 Structure of SHC and the colored boxes respectively standing for SPS, HLib and CStore.

4.1 | SHC Architecture

SHC consists of three major components: (1) Synchronization Processing Server (SPS), (2) Hooking Library (HLib), and (3) Customized IOStore (CStore)(shown in Fig. 7).

4.1.1 | SPS: Synchronization Processing Server

SPS is an independent process to maintain opened files for applications and ensure metadata consistency for them. As shown in Fig. 8, SPS is composed of the main thread, a thread pool, and several resource managers mainly including queues, maps, and vectors. The main thread is in charge of the `listen` socket function, serves as the only entrance for `connect()` and `send()` requests from clients, and responses the `accept()` and `receive()` operations. Upon receiving a `write()` request, the main thread creates a task and pushes the task into the tail of the task manager–a queue for the thread pool. Working threads in the thread pool stay being blocked if the task manager is empty. As the thread pool can only pop one task from the queue at a time, working threads have to compete for a task from the pool. We use two locks to maintain the mutual exclusions amongst threads. In addition, the locking design should satisfy the following demands since operations of a "queue" are not atomic operations:

1. The idle working threads should keep waiting when the main thread is pushing tasks;
2. The push operation should have higher priority than the pop;
3. All the idle working threads should be blocked when the task queue becomes empty.

The sPFS fd manager is a key-value table to maintain all the in-used sPFS files for client applications. It holds the working status and global file descriptor for an sPFS file. Each time an open request arrives, a new *fd* will be created and inserted into

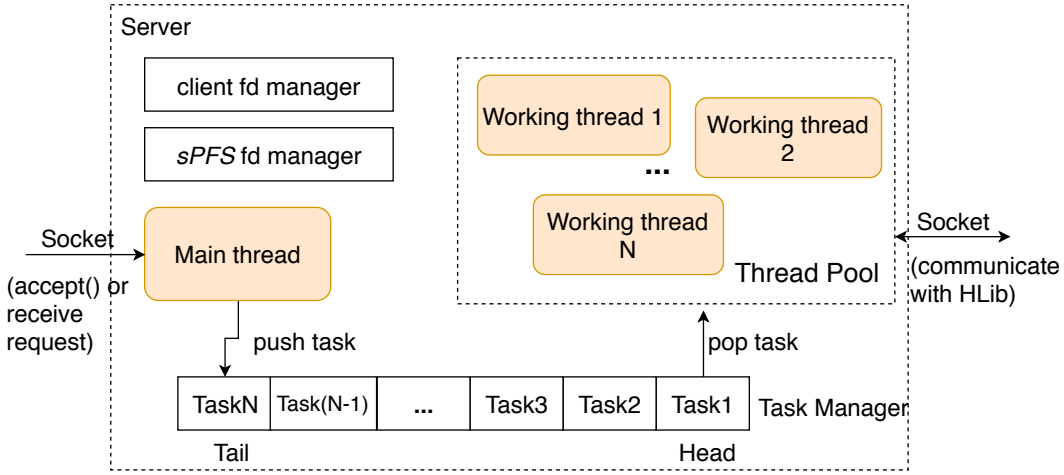

FIGURE 8 Structure of SPS. The rounded rectangles represent threads

the manager, unless a *fd* to that file has already existed. The existence of an *fd* indicates that an *sPFS* file is opened and is ready for writing data.

The client socket *fd* manager describes all the connected client processes. An instance of the client socket *fd* manager is comprised of a socket *fd* and a flag that indicates whether a working thread is processing a task that is sent from a client process. We stipulate that the messages from an active client should be received only by the assigned working thread in the server.

4.1.2 | HLib: Hooking Library

The Hooking Library (HLib) is a dynamic library linked to applications, which is based on the idea of *libplfs*. The HLib keeps the design of fundamental functionality that intercepts POSIX operations [19] and redirects them to another field. But differently, HLib will separate the processing of metadata and data by delivering operation commands to SPS with an empty data buffer but locally executing real I/O itself.

For writing requests, SPS will globally update the metadata for the target sPFS file and send HLib back a real I/O address referring to the backend of the sPFS. And for reading requests, the server will flush dirty metadata to underlying storage and reply an acknowledgment to HLib. Thus, HLib can pay no attention to the metadata consistency and only focus on executing real I/O operations. More details will be introduced in Section 4.2.

4.1.3 | CStore: Customized IOStore

IOStore identifies the module in PLFS that invokes the underlying file system interfaces(e.g., POSIX and PVFS) for actual data processing. I/O operations in PLFS will be finally delivered to the underlying file system through IOStores. For example, the POSIX IOStore will invoke POSIX interfaces to execute back-end I/O operations. And more generally, we adopt "IOStore" referring to an sPFS's module that interconnects the underlying file systems with the sPFS and executes real I/O operations.

CStore is a customed IOStore, which works in the server process and supports diverse underlying storage systems. When SPS tries to write data to the underlying file system, it is used to invoke an IOStore object with parameters including a real PLFS back-end address and a data buffer. As we mentioned in 4.1.2, the data buffer contains nothing but the socket fd describing where the front-end operations come from.

Instead of triggering actual data writing, CStore contacts the client process with that socket fd and transfer the back-end address to it for real I/O operation. In other words, when the server tries to execute a back-end writing operation, CStore will secretly cancel the writing action and "leakage" the real I/O address to the client with a socket fd hiding in the fake data buffer. More generally, CStore identifies the idea that modifying the IOStore of an sPFS to replace back-end writing with an address delivering to a client. During this procedure, a fake data buffer containing a socket fd supplies the information for the communication between the two.

4.2 | Data Stream

SHC contains an independent server process indicating that the data stream has inter-process communications(IPC). We choose sockets as the IPC technique among different nodes to assist SHC with the communications between the server process and application ones.

4.2.1 | Writes

The write synchronization issue in parallel file systems is the most important and complicated part of SHC. SHC handles write synchronizations by separating data writes and index write streams into different processes. Fig.9 demonstrates the complete write stream of SHC including communications and data streams:

First of all, HLib hooks a writing request which is originally sent to the kernel from an application. Then HLib rearranges the writing procedure if the target file path falls into one of *sPFS* mount points. HLib generates a socket for a file with the *pid* of the corresponding active process as its name. After connecting to the server socket, a message containing parameters of the write request such as data count, physical target path but an empty data buffer, is sent to the server process.

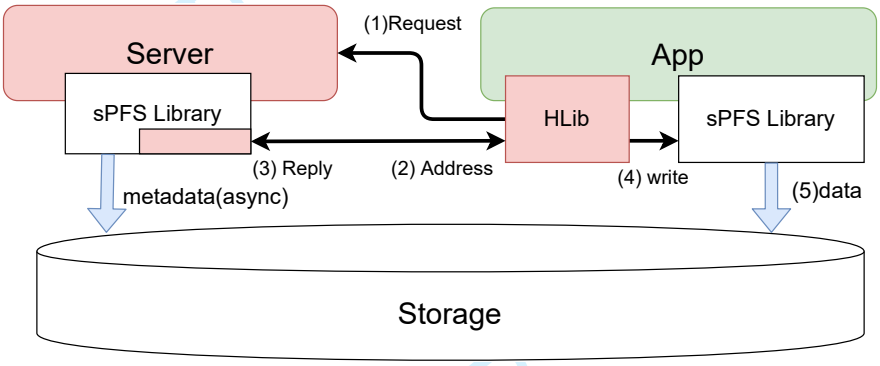

FIGURE 9 The procedure of writing a *sPFS* file

On the server side, the main thread accepts a connecting application from a client and receives a request. The received request is pushed into the task manager together with the client socket *fd* right away. A working thread then picks this task from the queue and parses a buffer for the task's parameters. The working thread calls *sPFS*'s `write()` after verifying a writing flag included in parameters. In particular, the respective client socket *fd* is delivered to *sPFS*'s writing function instead of a practical data buffer. Invoked by the writing function, the customized IOStore(a.k.a. CStore) obtain the socket *fd* and sends an address to the client with the *fd*. On the client-side, HLib writes data to the address that is provided by the server. This mechanism isolates the metadata and data operations.

SPS maintains a lazy mechanism to manage metadata for higher efficiency so that metadata will be pushed to underlying storage devices periodically rather than being flushed immediately when writes are issued. The writing isolation between data and metadata prevents writing requests from simply waiting in a queue for completely processing. And the design will introduce much fewer overheads than to simply line up the requests.

4.2.2 | Reads

The read operation is simpler than writes in SHC. Fig. 10 presents the data flow of communication and read data streams. The procedure of reading a *sPFS* file is similar to a common linking library in some way.

At first, HLib hooks a read request from an application and exams its path. HLib then connects and notifies the server process to flush the related metadata from memory to disks to keep the modified information updated on persistent storage. The server process will check the existence of a *fd* for such file in the *sPFS* manager and flush the metadata if a *fd* exists. Otherwise, the server process does not perform any flush operation. Notice that this interaction only occurs once during the whole reading action in spite of the number of `read()` is called. After receiving the reply from the server process, HLib invokes *sPFS* reading

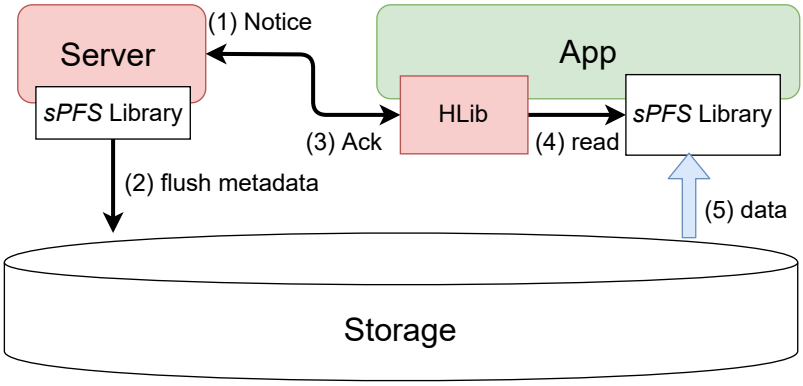

FIGURE 10 The procedure of reading a *sPFS* file

function for the target data. The main role of HLib is to maintain a mapping between a logical *sPFS* file and a *tmp* file which is held by an application.

4.3 | Security

Although we realize our conjecture of SHC, the proposed SHC implementation still steps far away from an effective alternative to FUSE. We propose some preliminary moves that we tried to push SHC one step closer to its security goal. We design a mechanism to manage connections between the server and client sockets under a limited connection quantity of server sockets. The mechanism is designed based on the following investigations:

- Keeping a client connected to the server all along is not efficient even in concurrent I/O circumstances.
- After opening a file, it must stay open in SPS to accept the successive write(read) operations until `close()` is invoked.
- Not all developers remember to invoke `close()` under heavy I/O workloads.

We design a mechanism on the server-side to manage numbers of client connections with an occupancy count for each client *fd*. An individual timestamp is set for each active client in the client socket *fd* manager when an action like receiving occurs. Every time the main thread updates the *fd* set before receiving a connection, the thread checks the inactive clients' timestamps and evacuates the client with the oldest timestamp to make room for other ones. In addition, those connected clients that have been idled long enough to trigger a threshold will be closed as well. As a client is disconnected, the related *sPFS* file that has a *fd* object in the PLS file manager will be closed.

Another design consideration lies in: after obtaining a request, the main thread does not parse the received buffer, while the working thread that chooses a task does have to parse the buffer. In this case, the main thread handles the connections and requests in a concurrent I/O circumstance in a more efficient way.

5 | SHC EVALUATION

5.1 | Experimental Configurations

Hardware: We deploy our SHC-PLFS on a Sunway TaihuLight HPC testbed cluster to test the I/O concurrency and up-scalability. Each node is equipped with two 2.6GHz 8-core 16-thread Intel Xeon CPU and 64GB RAM, running CentOS v7.5. The cluster is built on a Lustre file system with 1 MDS and 4 OSSes interconnecting with InfiniBand. Lustre's storage targets consist of 12 HDDs which are arranged as two groups of RAID6 by a Sugon DS800-F20 storage system providing 71TB HDD storage capacity.

Software: We implement PLFS (v2.5) with FUSE (v2.9.7) (denote as FUSE-PLFS), and compare the performance with SHC-PLFS obtained by *fs_test* benchmark[8]. We use *fs_test* to generate a scenario where a lot of data is modified concurrently to reflect the overhead of *open/sync/close* operations. We use Open-MPI (v3.1.0) to manage multiple processes in our tests.

Workloads: We use *fs_test* to evaluate the performance of SHC-PLFS system and FUSE-PLFS system. For each system, we try to explore its writing or reading capacities under different conditions and then compare them with each other. We set several workloads covering three important factors including the number of writing/reading threads, file size, and transfer size (see Table 2). Similar to the case study in Section 3.3, we warm up the page cache before running read tests and flush the memory after every write test.

TABLE 2 Summary of workloads

type	write, read
threads#	4,8,16
file size(MB)	4,16,64,256,1024
transfer size(KB)	32,64,128,256,1024

In addition to the ordinary Lustre, we compare the proposed SHC with three FUSE setups: default FUSE (denote as *FUSE-PLFS default*), FUSE with 128KB buffer (denote as *FUSE-PLFS w/128K buf*), and FUSE with direct I/O (denote as *FUSE-PLFS w/ DirectIO*). *FUSE-PLFS default* serves as the control group that FUSE buffer size is 4KB by default; *FUSE-PLFS w/128K buf* sets the buffer size to 128KB so that data smaller than 128KB can be transferred only once in FUSE kernel; *FUSE-PLFS w/ DirectIO* shares the same setup that is discussed in Section 3.3 to reveal the impact of page caches.

5.2 | Benchmark Evaluation and Analysis

5.2.1 | I/O Bandwidth

Fig. 11 shows the *fs_test* results of writing and reading bandwidth under different conditions which are file size, transfer size and number of processes. In Fig. 11, SHC-PLFS shows outstanding performance compared with the FUSE-PLFS system.

Transfer sizes: Transfer size is one of the most important factors of the performance. For a fixed amount of data, a smaller transfer size means a larger number of I/O operations(nobj). Since SHC has to maintain communications between applications and SPS through sockets for each I/O operation, the smaller the transfer size is, the larger proportion the extra overheads caused by sockets will take in the whole performance. In this section, we present comparisons of bandwidth between SHC-PLFS and FUSE-PLFS with changes in transfer sizes. Fig. 11(a) shows sharply increasing write bandwidth of SHC with the growth of transfer size. Differently, the read bandwidth of SHC stays stably on a leading level because of the page cache and the *readahead* mechanism of Linux.

We notice that SHC-PLFS presents a great superiority compared with FUSE-PLFS. One of the main reasons is that FUSE-PLFS produces more than one memory copy, while SHC does not introduce extra data copy between kernel and user-space. Besides, additional context switches and complex mechanisms limit the behaviors of FUSE-PLFS cases.

In the read case, FUSE-PLFS presents stable performance for the same reason as SHC-PLFS. In particular, FUSE-PLFS using *direct IO* gives up the page cache for efficient write performance which results in poor reading behavior with small transfer sizes. But it catches up with SHC-PLFS when facing large transfer sizes probably because of the local cache on Lustre's OSS(Object Storage Server). Lustre, the underlying file system we use in this evaluation, arranges data cache on its OSSs to avoid frequently accessing OSTs(Object Storage Target).

Lustre performs a stable level of bandwidth around the tests as it is not really good at handling the N-1 pattern. Internally, Lustre stripes a file into a specific number of objects each of which will be stored on an OST. Although Lustre can handle data accessing in parallel by striping, data writing on the same file object still triggers lock competitions. Differently, PLFS changes N-1 into N-N I/O pattern and processes do not need to compete for the same object on an OST.

File sizes: In Fig.11(b)(e), the competitors show increasing trend with ascending file sizes except Lustre. We choose 64KB as the transfer size in this test since large sizes show the great disparity between FUSE-PLFS and SHC-PLFS while small sizes show the inferiority of FUSE-PLFS using direct I/O in Fig.11(a)(b). In write cases, FUSE-PLFS using direct I/O shows better performance than the other two FUSE cases, as it directly maps the application's data to FUSE daemon without copying data to FUSE's cache in the kernel. Both SHC-PLFS and FUSE-PLFS using direct I/O keep bandwidth increasing with the growth of file sizes. Because the latency of handling file metadata and interconnection within Lustre such as lock competition will take less proportion in the whole test. In read cases, FUSE-PLFS with direct I/O surpasses the other two FUSE-PLFS cases when

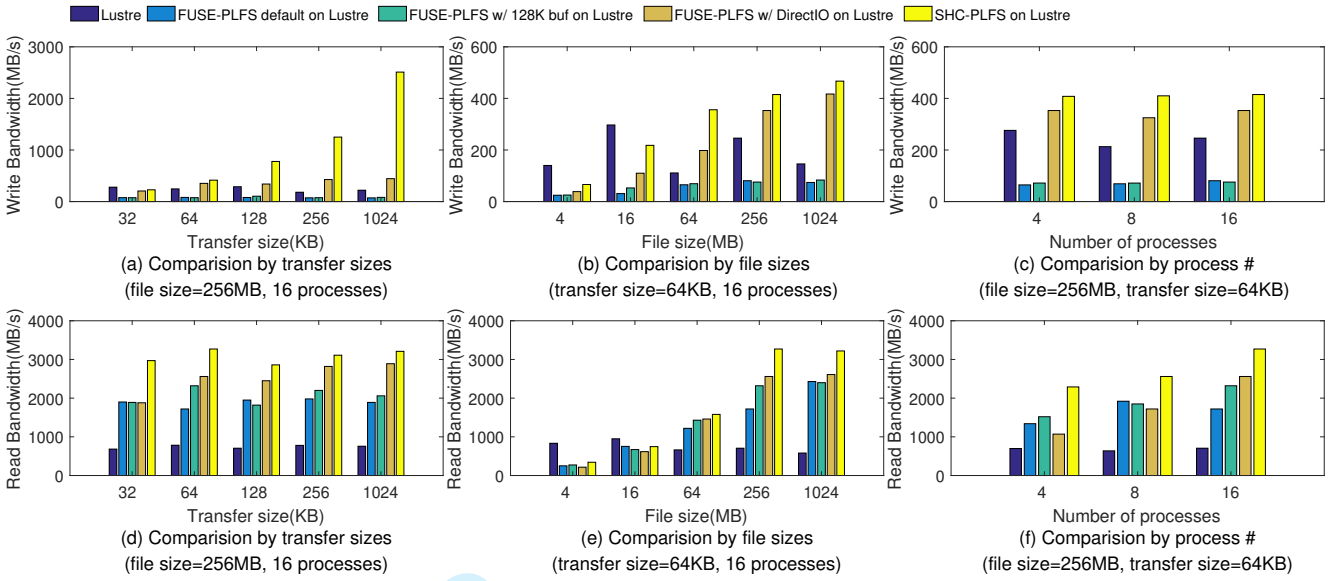

FIGURE 11 Comparisons on writing(a-c) and reading(d-f) bandwidth. (a) and (d) show the bandwidth with different transfer sizes when 16 processes concurrently read/write a 256MB file. (b) and (e) present the increasing trend of SHC with the growth of file sizes under 16 processes. And (c) and (f) gives the effect of different numbers of processes.

reading large files although they can benefit from the reading cache in FUSE. Nevertheless, SHC-PLFS leads the performance since it benefits from both the reading cache and less data copying.

Processes Quantity: Fig. 11(c) and (f) show the comparison among the five competitors under different numbers of processes. In write cases, *FUSE-PLFS w/ DirectIO* and SHC-PLFS both present outstanding performance while the other two FUSE cases are still burdened with memory copy in the FUSE kernel. But in reading cases, although memory copy affects both *FUSE-PLFS default* and *FUSE-PLFS w/128K buf*, page cache and readahead mechanism help them catch up with *FUSE-PLFS w/ DirectIO*.

5.2.2 | Open and close time

Open time results of the two systems are shown by Fig. 12. SHC keeps staying at a stable low level of open latency, while open times of both *FUSE-PLFS default* and *FUSE-PLFS w/128K buf* oppositely present much higher. The open time of FUSE-PLFS shows a sharp increment with the growth of not only file size but also a number of processes. When opening a file, FUSE needs to create a corresponding handle in its kernel module before delivering the `open()` request to PLFS[7]. There may be a number of lock competitions among queues in FUSE's kernel module. More importantly, FUSE has to flush its kernel cache before each open operation to ensure data consistency, which impacts the open time seriously. And large file size means cache flushing may take longer before open(). In contrast, all of SHC-PLFS, Lustre, and *FUSE-PLFS w/ DirectIO* do not need to worry about these overheads. Thus, they present extremely low open time compared to both *FUSE-PLFS default* and *FUSE-PLFS w/128K buf* with the growth of the file size.

However, the close time of SHC-PLFS and the three FUSE cases fluctuate irregularly and show serious inferiority compared to Lustre. Because PLFS has to flush all the related caching data and update lazy metadata to the underlying storage when closing a file. Thus, their latency is far longer than Lustre.

5.3 | Comparison with LDPLFS

As we mention in Section 3, LDPLFS is attached to the application process and can not communicate with other processes. Without using MPI-IO which is seldom used outside the HPC center, LDPLFS can not handle concurrency which may cause data consistency problems. We discuss their tight coupling in Section 3.

Generally, LDPLFS works well for data reads. When it comes to concurrent writes to a single namespace, it is another story. If more than one processes try to write to the same file, multiple writes may not be issued on the correct locations and

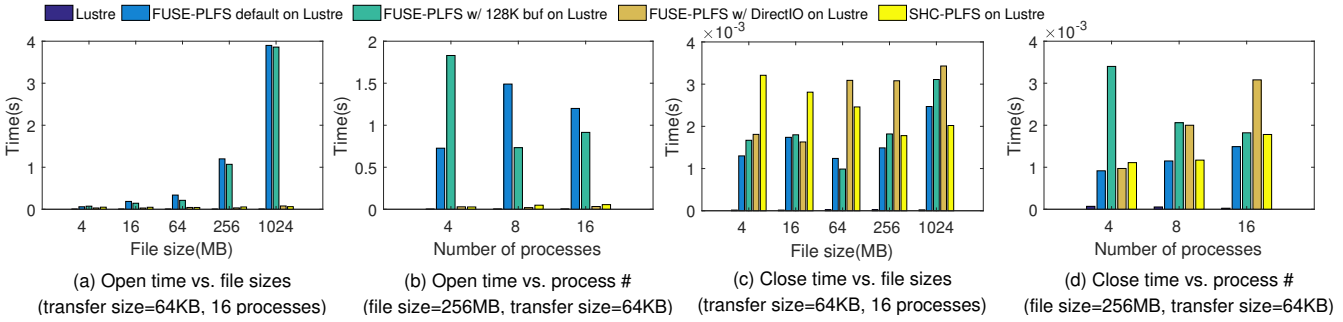

FIGURE 12 (a) and (b) are comparisons of open time between SHC-PLFS and FUSE-PLFS; (c) and (d) are related to close time.

may introduce data overlapping. Such communications amongst the applications can be achieved via FUSE, this mechanism is eliminated when FUSE is bypassed. This could lead to a severe data inconsistency problem. Furthermore, LDPLFS does not take care of the miscalling `close()` in applications. In this case, any lazy I/O operations may lead to a problem that metadata would not be persistently stored as the consequent `plfs_close()` will not be invoked before a process exits.

Even though LDPLFS does not suit scenarios without MPI-IO, it gives the upper bound of SHC's performance since SHC is based on the preload library method. And Wright *et al.* has compared LDPLFS with PLFS deployed with MPI-IO and LDPLFS showed shortages in the performance. So that we do not need to compare SHC with the PLFS with MPI-IO.

Thus, we run additional tests on a 9-node cluster to compare SHC's performance with LDPLFS. The cluster is equipped with Intel Xeon E5-2603 and 16GB DDR4 memory for each node and runs on a PVFS file system. Fig.13 compares performance between LDPLFS and SHC. The results are the average of the 5 tests and the result errors of the six cases range between 2% and 8%. As we anticipate, LDPLFS *hooking* method shows an outstanding performance. We can observe that SHC retains a similar read performance compared to LDPLFS. The little advantage is within the error range. But SHC can reach 65-96% write performance of LDPLFS, which is mainly due to the SHC additional metadata management. Even so, SHC shows an acceptable performance on both reading and writing.

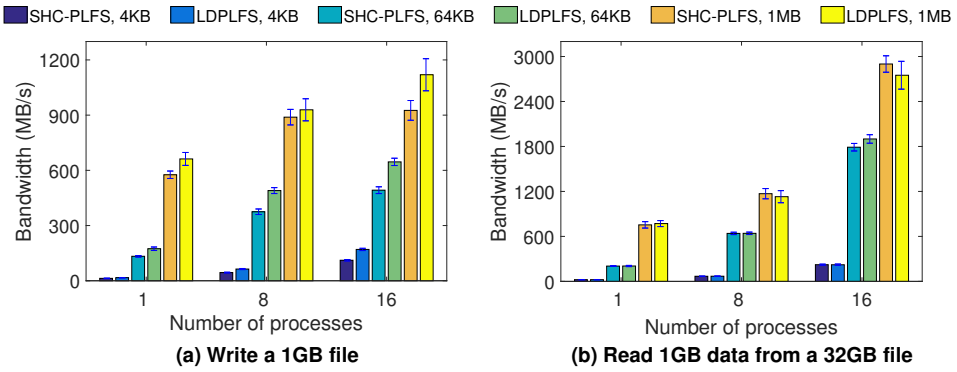

FIGURE 13 Both writing and reading comparisons are under a different number of processes and transfer sizes. (a) Each data block is synchronized after executing `write()`; (b) We release cache between two workloads.

We present a small experiment to study the potential deficiency of LDPLFS. An OrangeFS (PVFS) is running on top of the cluster that consists of four compute nodes and four data servers. The total storage capacity of the cluster is 4TB. We fork four processes to emulate four independent applications and make them write to the same file in the PLFS frontend. The file is written in an append mode. Table 3 shows the recorded file size via `stat()` function under the three writing methods (FUSE, SHC, and LDPLFS). We notice that LDPLFS presents only a quarter amount of data from the frontend while storing all four pieces of data at the backend. This indicates that LDPLFS treats the four writes independently and mis-updates the metadata due to the lack of synchronizations.

TABLE 3 File sizes(KB) after writing by four processes

per process	total size	FUSE	SHC	LDPLFS
64	256	256	256	64
256	1024	1024	1024	256
1024	4096	4096	4096	1024

In addition to the misrepresenting data sizes, we design another small experiment to investigate the missing `close()` scenario. We create a process to open a file, write data, force it to exit without invoking `close()`, and examine the corresponding metadata. We observe that the metadata of the file is NULL while the written data is stored at the backend. The major reason is that without FUSE intervention, kernels may not be notified to flush the metadata only if applications raise explicit `close()` calls.

5.4 | Evaluation on A Real World Application

We further evaluate SHC with the Beacon system which is an I/O monitoring system of TaihuLight Supercomputer. Beacon is developed under POSIX standard and started to monitor the TaihuLight supercomputing machine in 2017. One of the main operations of Beacon is to collect I/O characteristics of TaihuLight and asynchronously flush them to a Lustre cluster every minute with a large number of log files. The log files are then integrated into one file asynchronously on Lustre. The periodical file integration becomes an increasing burden of Beacon as compute clusters go up-scaling. Beacon is trying to handle this problem by aggregating discrete logs into a shared file during the collection of I/O characteristics. In this way, Beacon can reduce redundant labor for integration after it flushes the logs to Lustre. However, Lustre intends to achieve higher I/O bandwidth by accessing large-sized files, which may not fit Beacon's I/O patterns where multiple log files are aggregated into one file (a.k.a. N-1 I/O pattern). With the assistantship of SHC, Beacon can directly work with PLFS and take benefit of PLFS in handling N-1 I/O accesses without any code modification.

Thus, we integrated an SHC prototype in Beacon and implemented it on a TaihuLight testbed system. The Beacon system we are testing consists of four servers, each of which has two processes. The data for the experimental test is collected by Beacon for four days on a TaihuLight subsystem with 4096 compute nodes. The data size is 39GB, 35GB, 20GB, and 32GB for each day correspondingly. Fig. 14 demonstrates the data write time comparison of three cases: 1. Beacon's current data management methodology where multiple log files are written to Lustre then aggregated into one file (denote as "**Beacon**"), 2. Beacon's attempt to write log files to a shared file concurrently while Lustre is in charge of data consistency (denote as "**Beacon-SF**") 3. Beacon that integrated with SHC mechanism (denote as "**Beacon-SHC**").

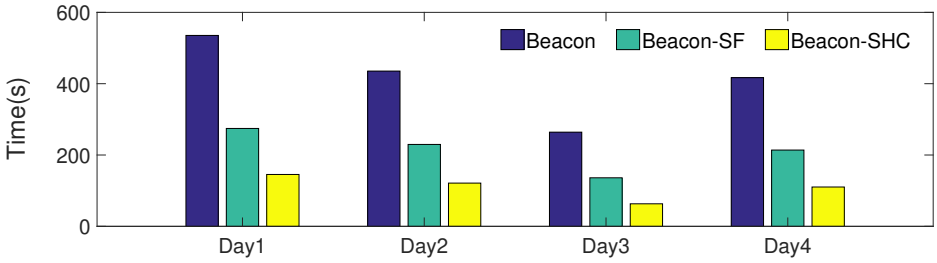

FIGURE 14 Data write time comparison

We can observe from Fig. 14 that SHC-assisted Beacon ("**Beacon-SHC**") outperforms the existing Beacon solution mainly because SHC simplifies the data processing routines by combining the data writing and aggregation into a single step. We also find that Beacon-SHC performs 1x quicker than Beacon-SF, where Beacon uses Lustre to maintain data consistency while it writes multiple log files into one shared file. The Lustre Distributed Lock Management mechanism (a.k.a. LDLM) introduces a significant overhead towards metadata when Beacon tries to write to a shared file.

6 | DISCUSSION

Scalability: The SHC introduces an SPS as a centralized server to guarantee data consistency. Although SPS handles only a part of metadata operations, it will inevitably fall into scalability problems since there is only one server currently. It should be a key challenge towards the upscaling of computer systems. But it could be solved by improving the design of SPS for scalability, which is like the development of some existing parallel file systems. For example, early Lustre[21] had only one metadata server (MDS) and it nowadays can serve multiple MDS after implementing the Distributed Name Space (DNE) technique on it. SPS can theoretically use the similar idea of techniques like DNE to extend the number of servers. Thus, it could be future work to solve the scalability problem.

MPI-IO: MPI-IO provides a low-level interface to carry out parallel I/O. There barely have tools to analyze what kind of data is stored in the file using MPI-IO API[22]. Data consistency could be a problem if we transform API from POSIX to MPI-IO using LD_PRELOAD. MPI-IO relaxes POSIX semantics and defines an interface that allows applications to manage cache coherency themselves. Even though the MPI_FILE_SYNC command gives applications a guarantee about the freshness of data by controlling when data and metadata are flushed and re-validated, this mechanism can not be replaced by LD_PRELOAD automatically. What's more, the data consistency by MPI_FILE_SYNC is restrained within a process. When it comes to data consistency across multiple processes, the command may not be competent. SHC, on the other hand, extends the functionality of local consistency under POSIX and maintains data consistency across multiple processes. It can be a more considerable solution for regular users.

Stack-ability: Different from a common file system, stackable file systems lack kernel components so that developers can be more succinct to achieve their parallel ideas. Although stackable file systems generally rely on some frameworks like FUSE, the number of parallel file system products would meet a swift growth if extra overheads caused by the framework can be a restraint to a lower level.

The performance gain in our experiment with FUSE exempted proved that FUSE does have drawbacks, and could be a bottleneck, especially on parallel file systems. When developing and deploying parallel file systems, the tradeoff upon whether to put the system in user space or kernel space should be considered. A reliable file system kernel module is certainly more efficient than a user-space library. As it is in kernel space, it is more efficient to do memory management and utilize cache coherency. However, many parallel file systems do not have an official in-kernel deployment option, which means only FUSE or a proprietary API is available to the developer, introducing a significant overhead on developing, debugging, and deploying time and costs.

We believe FUSE is usually not an optimal solution for deploying parallel file systems if FUSE is not fine-tuned to fit the file system design. First, parallel file systems are usually large-scale and capable of highly concurrent I/Os. The file system itself could handle concurrent I/Os very well, but FUSE could introduce an overhead here because of the internal queuing or serialization. Second, stackable parallel file systems are more like a file organizer, but not an actual file keeper, thus all the organizing jobs could be done in user-space without any kernel crossings before the final underlying operation. An interposition library like *libplfs* is well suited for the job while FUSE is too heavyweight. Finally, the isolation between user space and kernel space is firmer than ever thus it is time to consider reducing system call frequencies by moving some non-kernel jobs to the user-space.

7 | RELATED WORK

File System Middleware: There are several I/O middleware layers currently developed for HPC environments. Reaching exascale I/O performance tends to rely on these middleware layers capable of managing parallel I/O workloads. Work has been conducted on matching the user view of parallel I/O to optimize workloads on a parallel file system, but PLFS takes this further and attempts to mask I/O workload and system configuration parameters from users. Similar to the PLFS project, the Adaptable I/O System (ADIOS) from Oak Ridge National Laboratory is an I/O library and API for scientific codes that efficiently groups scientific array data and is capable of writing the data in a log-structured format[23]. The Distributed Application Object Storage (DAOS) from Sandia National Laboratory serves as the persistent storage interface and translation layer between the user-visible object model and the requirements of the underlying storage infrastructure[24]. Besides, the RAMCloud Storage System from Stanford University offers low-latency data services by aggregating the main memory of thousands of servers for a single coherent key-value store[25]. These projects provided file system deployment and optimization ideas.

User-level File Systems: There are many industrial level or academic use file systems that reside in user space, like Wayback[26], ChunkFS[27], HydraFS[28] and DeltaFS[29]. Many popular parallel and distributed file systems also provide an option to be mounted via FUSE, like PLFS[4], CephFS[20] and PVFS[30]. There are also several researchers designed extensions for FUSE[16,31,32,33]. FUSE is such a common mechanism in developing new file systems for distributed computing and high-performance computing domains. Due to the overhead that FUSE introduced, developers have to balance the simplicity and performance overhead by FUSE. Vangoor et al. proposed a detailed discussion of the architecture of FUSE and its pros and cons[7]. The study further compares FUSE's performance with different workloads in various aspects such as handling metadata, handling I/O, and managing diverse devices. The wide usage of FUSE motivates us for further research on it.

Parallel File Systems: Parallel file systems have a critical position in high-performance computing. We have mentioned many parallel file systems in our paper. There are many popular research parallel file systems like Galley[34], PPFS[35] and PIOUS[36]. These parallel file systems usually have experienced decades growing and some were widely deployed in HPC data centers. Some modern high performance distributed or parallel file systems are, for example, Lustre[1], CephFS[20] and OrangeFS[37]. The aforementioned parallel file systems have internal mechanisms to ensure data consistency via one or more metadata servers. Parallel file systems usually maintain a distributed lock mechanism to ensure metadata consistency. Operations from applications then are uniformly delivered to clients via the POSIX interface and performed locally. Such a centralized processing mechanism avoids the contention problem amongst applications.

Dynamic library: Dynamic library offer functionalities for applications without modifying their source code. LDPLFS[19] proposed a dynamic linking library to eliminate additional kernel accesses which are introduced by FUSE. However, LDPLFS may lead to writing inconsistency issues if applications do not take care of the synchronization problem. Other dynamic linking cases are similarly designed to assist specific stackable file systems–TableFS[38] and FusionFS[39], for example–to offer another implementation improving performance or achieve additional functionality. Direct-Fuse is a framework aiming at bypassing FUSE for general applications[40]. Direct-Fuse shares a similar idea of the dynamic library, but it is more application-dominated. The major problem of existing dynamic linking library mechanisms is that the library is determined by application processes in the user space, which makes communications amongst applications difficult.

8 | CONCLUSION AND FUTURE WORK

In the paper, we proposed an implementation scheme for a user-level file system to improve its performance as well as solve the consistency problem. We study the overheads of FUSE which is a commonly used framework to deploy a user-level file system and find an alternative implementation for the user-level file system. Our scheme helps the user-level file system to get rid of FUSE with a dynamic library, and introduces a lightweight server to solve the data consistency problem caused by the lack of FUSE.

We first conduct a case study to examine the effects of FUSE on the performance of user-level file systems. Then we implement a preliminary solution for "PLFS without FUSE" using a dynamic library on CephFS. We then present a mechanism called SHC to reduce kernel crossings of a parallel file system in user-space based on the discussion of the limitation of FUSE when mounting user-level parallel file systems. The SHC is then applied to PLFS and is further implemented on an 8-node Sunway TaihuLight HPC testbed Lustre file system with InfiniBand connections. *fs_test* benchmark testing results indicate that kernel crossings become a major I/O performance bottleneck in a concurrent I/O circumstance for user-level parallel file systems.

The current SHC has its limits in handling scalability, which should be a key challenge towards computer systems upscaling. We will address this carefully in future work. As an ongoing project, we will make efforts to introduce redundancy to the centralized server. We will also handle the scalability by applying a dedicated cluster and maintaining metadata parallelly, similar to Lustre's distributed name-space technique.

ACKNOWLEDGMENTS

Shu Yin's research is supported by the China Postdoctoral Science Foundation under Grant 2015M572708, and ShanghaiTech University under a start-up grant. Xiaomin Zhu's work is supported by the National Natural Science Foundation of China under Grant 61572511, and the Scientific Research Project of National University of Defense Technology under Grant ZL16-03-09.

References

1. Cluster File Systems I. Lustre: A Scalable, High-Performance File System. 2002.
2. Schmuck F, Haskin R. GPFS: A Shared-Disk File System for Large Computing Clusters. In: FAST '02. USENIX Association; 2002; Berkeley, CA, USA.
3. Edge J. The OrangeFS distributed filesystem. 2015.
4. Bent J, Gibson G, Grider G, et al. PLFS: A Checkpoint Filesystem for Parallel Applications. In: ; 2009
5. Wikipedia . Filesystem in Userspace. 2018.
6. Tarasov V, Gupta A, Sourav K, Trehan S, Zadok E. Terra Incognita: On the Practicality of User-Space File Systems. In: USENIX Association; 2015; Santa Clara, CA.
7. Vangoor BKR, Tarasov V, Zadok E. To FUSE or Not to FUSE: Performance of User-Space File Systems. In: USENIX Association; 2017; Santa Clara, CA: 59–72.
8. LANL . fs_test. 2017.
9. Lipp M, Schwarz M, Gruss D, et al. Meltdown. *ArXiv e-prints* 2018.
10. Chen C, Liu J, Zou Y, Deng T, Zhu X, Yin S. A Case Study on the Efficiency of User-Level Parallel File Systems. In: ; 2019: 90-97
11. Zou Y, Chen C, Deng T, et al. SHC: A Method for Stackable Parallel File Systems in Userspace. In: ; 2019: 1374-1381
12. User:Sven . FUSE structure(wiki). 2008.
13. Hoskins ME. Sshfs: super easy file access over ssh. *Linux Journal* 2006; 2006(146): 4.
14. Davies A, Orsaria A. Scale out with GlusterFS. *Linux Journal* 2013; 2013(235): 1.
15. Ouyang X, Rajachandrasekar R, Besseron X, Wang H, Huang J, Panda DK. CRFS: A lightweight user-level filesystem for generic checkpoint/restart. In: IEEE. ; 2011: 375–384.
16. Ishiguro S, Murakami J, Oyama Y, Tatebe O. Optimizing local file accesses for FUSE-based distributed storage. In: IEEE. ; 2012: 760–765.
17. Yang B, Ji X, Ma X, et al. End-to-end I/O Monitoring on a Leading Supercomputer. In: ; 2019; Boston, MA: 379–394.
18. IOR:Parallel filesystem I/O benchmark. 2011. https://github.com/LLNL/ior.
19. Wright SA, Hammond SD, Pennycook SJ, Miller I, Herdman JA, Jarvis SA. LDPLFS: Improving I/O Performance without Application Modification. In: ; 2012: 1352-1359
20. Weil SA, Brandt SA, Miller EL, Long DDE, Maltzahn C. Ceph: A Scalable, High-performance Distributed File System. In: OSDI '06. USENIX Association; 2006; Berkeley, CA, USA: 307–320.
21. Wikipedia . Lustre (file system). 2018.
22. Gropp W, Lusk E, Thakur R. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press . 1999
23. Lofstead JF, Klasky S, Schwan K, Podhorszki N, Jin C. Flexible io and integration for scientific codes through the adaptable io system (adios). In: ACM. ; 2008: 15–24.
24. Lofstead J, Jimenez I, Maltzahn C, Koziol Q, Bent J, Barton E. DAOS and friends: a proposal for an exascale storage system. In: IEEE Press. ; 2016: 50.

25. Ousterhout J, Gopalan A, Gupta A, et al. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 2015; 33(3). doi: 10.1145/2806887

26. Cornell B, Dinda PA, Bustamante FE. Wayback: A user-level versioning file system for linux. In: ; 2004: 19–28.

27. Henson V, Ven v. dA, Gud A, Brown Z. Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair.. In: ; 2006.

28. Ungureanu C, Atkin B, Aranya A, et al. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System.. In: . 10. ; 2010: 225–239.

29. Carns P, Lang S, Ross R, Vilayannur M, Kunkel J, Ludwig T. Small-file access in parallel file systems. In: ; 2009: 1-11

30. Carns PH, III WBL, Ross RB, Thakur R. PVFS: A Parallel File System for Linux Clusters. In: USENIX Association; 2000; Atlanta, GA.

31. Narayan S, Mehta RK, Chandy JA. User space storage system stack modules with file level control. In: Citeseer. ; 2010: 189–196.

32. Sundararaman S, Visampalli L, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Refuse to crash with Re-FUSE. In: ACM. ; 2011: 77–90.

33. Rajgarhia A, Gehani A. Performance and Extension of User Space File Systems. In: SAC '10. ACM; 2010; New York, NY, USA: 206–213

34. Nieuwejaar N, Kotz D. The Galley parallel file system. *Parallel Computing* 1997; 23(4-5): 447–476.

35. Huber Jr JV, Chien AA, Elford CL, Blumenthal DS, Reed DA. PPFS: A high performance portable parallel file system. In: ACM. ; 1995: 385–394.

36. Moyer SA, Sunderam V. PIOUS: a scalable parallel I/O system for distributed computing environments. In: IEEE. ; 1994: 71–78.

37. Bonnie MMD, Ligon B, Marshall M, et al. OrangeFS: Advancing PVFS. 2011.

38. Ren K, Gibson G. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In: USENIX; 2013; San Jose, CA: 145–156.

39. Zhao D, Zhang Z, Zhou X, et al. FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: ; 2015

40. Wang T, Mohror K, Moody A. Direct-FUSE : Removing the Middleman for High-Performance FUSE File System Support. 2018.