2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

SHC: A Method for Stackable Parallel File Systems in Userspace

Yanliang Zou[†], Chen Chen[†], Tongliang Deng[†], Jian Zhang[†], Si Chen[¶], Xiaomin Zhu[||], and Shu Yin[*†||]
[†]School of Information Science and Technology, ShanghaiTech University, China, Email: yinshu@shanghaitech.edu.cn
[¶]West Chester University, PA, USA
[||]National University of Defense Technology, China
[*]Corresponding Author

***Abstract*—We present SHC, a stackable layer that leverages the power and flexibility of middlewares to handle kernel crossings of parallel file systems in userspace. The core of SHC is a dynamic library interception mechanism that improves I/O performance and guarantees consistency. Additionally, our solution uses a server process to handle write synchronizations so as to maintain the data correctness and the data consistency with minimal overheads. We implement an SHC prototype with PLFS on a Lustre system and demonstrate that SHC maintains most interactions between applications and PLFS in userspace while requiring fewer kernel calls. We show that SHC improves write bandwidth up to 30x higher than FUSE, and reduces the latency of up to 90% for high-performance I/O systems.**

***Index Terms*—Stackable Parallel File Systems, PLFS, FUSE, I/O performance, Kernel Crossing**

I. Introduction

Parallel file systems provide a single-node massive storage abstraction and distributing files in a striped manner. In order to reduce kernel programming difficulties, the design of new parallel file systems usually seek help from user space file systems like FUSE (a.k.a. Filesystems in Userspace) [1], which acts as an agent to assist file systems to be mounted in the userspace. The FUSE technique relieves developing workloads for new file systems, however, it introduces performance overheads under data-intensive and concurrent I/O scenarios due to the additional cache accesses and memory copying redundancies between the user mode and the kernel mode. Serving as an agent that transfers an application's requests in the userspace to underlying stackable file systems for execution, FUSE behaves like a separate application to the kernel which leads to a duplicate data cache and twice as much data transfers between the kernel and the userspace.There are at least two data transfers when FUSE is involved – one occurs between an application and FUSE kernel while the other one is between FUSE-based stackable file systems and the underlying file systems. According to Vangoor and Zadok, additional data transferring will occur between the FUSE kernel and FUSE-based stackable file systems if Splicing technique is not supported [2].

In order to retain lower development difficulties for new parallel file systems while avoiding manipulations of existing applications, we propose a dynamic library approach that bypasses the FUSE without additional modifications to applications. Although dynamic library approaches have been commonly studied, such as TableFS [3] and FusionFS [4], data consistency may not be maintained if synchronization mechanisms are not guaranteed. Take the IoT network as an example where multiple sensors may write to the same file without communicating with each other, the dynamic library itself may lead to data errors if synchronization is not well addressed. Our customized dynamic library hooks requests from users and redirects them to the underlying file systems by overriding general system calls under POSIX. We called this method SHC which is comprising three key components: Server Process for Synchronization(SPS), Hooking Library(HLib) and Customed IOStore(CStore).

SPS is to deal with the write synchronization issue when multiple processes are trying to write to the same file by segmenting. It maintains data consistency but introducing no additional memory copying and data buffering.

HLib is a library to capture I/O requests and responses for determining where they should be delivered to. It is linked to an application where the requests come from.

CStore is a customized version of IOStores of the FUSE-based stackable file systems library, which is a handle to utilize underlying file system interfaces such as POSIX and PVFS. This modified library is linked to **SPS** and coordinates data writing with client processes.

The proposed SHC aims at 1) avoiding the FUSE intervene for stackable file systems; 2) avoiding additional modifications to applications; 3) maintaining the data consistency when multiple users write to a single file without communicating each other, and 4) achieving acceptable systems performance compared to systems with FUSE.

As for the server and client processes of SPS, we utilize the socket that supports multi-connection to achieve interprocess communication. Even though the socket mechanism introduces communication overheads amongst server and client processes, I/O systems still can gain benefits from the reduction of memory copy operations and of redundant buffered data.

Our contributions are summarized as follows:

- We present SHC, a method to ensure the implementation of stackable parallel file systems in userspace while introducing fewer overheads than that of FUSE solutions;
- We implement an SHC prototype on PLFS [5] and run tests on the eight-node testbed cluster at Sunway TaihuLight.

978-1-7281-2058-4/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2019.00191

- SHC presents a method to improve the up-scalability of the userspace parallel file systems by eliminating the bottleneck of FUSE.

The remainder of the paper is arranged as five parts: Section II provides the background and related work while Section III introduces the design of the proposed SHC. Section III also introduces the way to apply SHC on PLFS. Section IV evaluates the experimental results that are collected from a nine-node lab cluster and a testbed cluster at TaihuLight supercomputing center. Section V concludes the paper with future work.

II. Background and Related Work

Stackable parallel file systems (hereafter referred to $sPFS$) are those run in user space and exactly an application for the kernel. They realize the parallel logic and file system function as a completed parallel file system does. Normally, stackable parallel file systems are mounted on some user space file system frameworks whose representative is FUSE.

A. Background

FUSE (Filesystem in Userspace) is one of the most widely used frameworks in both academia and industry. An examination illustrates that at least 100 file systems are developed based on FUSE in recent years [2], [6]. FUSE consists of a kernel module and a group of interfaces in userspace. When mounting on FUSE, a file system inflects its behaviors to FUSE's interfaces so that it does not have to face the OS kernel. Requests from applications will be first sent to FUSE's kernel module; then they are scheduled to deliver to the target file system(*Figure 1*). Some researchers [2] analyze structures in FUSE's kernel level detailedly. In the FUSE kernel module, requests from applications will be inserted into different queues. By scheduling requests in these queues and sending them to its daemon running in userspace, FUSE realizes a reasonable mechanism to process tasks from different applications in user space.

However, FUSE introduces some unavoidable overhead for the stackable file systems, for example, double caching for the same data. As what we describe above, suppose a read request sent by an application reaches $sPFS$ through FUSE's transforming, both the underlying file system and FUSE will cache a duplicate of data respectively, which wastes half of the relevant memory and suffers lower performance when multi I/O for certain blocks occur in the data-intensive case.

Beyond that, additional times of memory copy are also a considerable problem in the concurrent environment. As what we introduce above, data for reading/writing requests have to experience at least two memory copying: 1)between FUSE and application; 2)between $sPFS$ and underlying file system. In addition, setting off splicing or old version of FUSE without splicing will cause another memory copying between $sPFS$ and FUSE, because splicing is a technique to create a direct memory mapping between FUSE and its mounting file systems instead of a necessary memory copying.

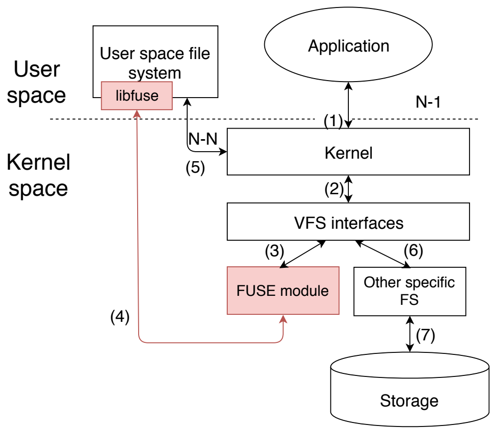

Fig. 1. The architecture and data flow of PLFS mounting on FUSE: Requests reach FUSE kernel module through VFS; then they are sent to PLFS in userspace. During this procedure, FUSE daemon acts as a dispatching station which response to receive and deliver requests to target file systems

Kumar *et al.*, Aditya *et al.*, Tarasov *et al.* [2], [7]–[10] discuss the FUSE's effect to file systems on performance but to my best knowledge, there is no research focusing on concurrent I/O scenarios.

B. Related work

Stackable file system is generally known as an abstraction layer between VFS and native file systems [11]. It offers functions as common file systems do but it does not realize actual I/O to devices which are done by the native file system it stacks on. Such a scheme decrease the complexity of developing a typical stackable files include Wrapfs [12], Cryptfs [13], SynergyFS [14] and etc. However, such a layer still has to develop the kernel module to mount on VFS and it would be much complicated to realize a parallel file system in the kernel environment. Therefore, thoroughly stripping stackable file system off from the kernel is meaningful. Nevertheless, it comes other problems without support from kernel such as cache coherency [15] and synchronization problem for a stackable parallel file system. User-space file system frameworks like FUSE comes out and serve as a solution [3], [16]–[18]. In this paper, we present SHC as another choice for a stackable file system in userspace.

Dynamic library is a software module shared by different applications in the OS. Before an application is run, the dynamic library will be linked to it and loaded into the memory. 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 [3] and FusionFS [4], for example–to offer another implementation improving performance or achieve additional functionality.

Direct-Fuse is a framework aiming at bypassing FUSE for general applications [20]. 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-level, which makes communications amongst applications difficult.

III. Design of SHC

In this section, we present the design of the proposed SHC idea. SHC mainly aims at mounting user-level stackable parallel file systems without FUSE intervene. 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 (e.g. IoT sensor network servers).

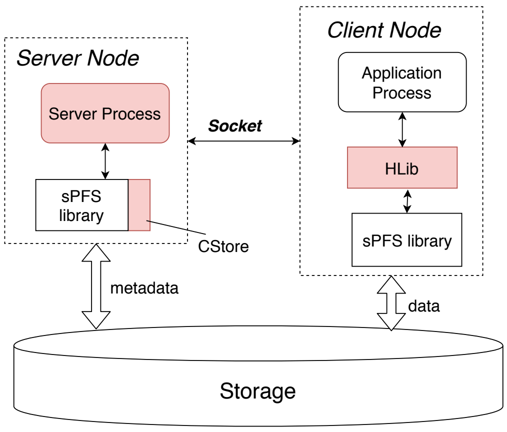

Fig. 2. Structure of SHC and the colored boxes respectively standing for SPS, HLib and CStore.

A. SHC Architecture

SHC consists of three major components: (1) Server Process for Synchronization(SPS), (2) Hooking Library(HLib), and (3) Customized IOStore(CStore)(shown in *Figure 2*).

1) SPS: Server Process for Synchronization: SPS is an independent process to deal with applications' writing requests. As shown in *Figure 3*, SPS is composed of the main thread, a thread pool, and several resource managers that are Standard Template Library(STL) containers. The resource managers in SPS mainly utilize three types of STL containers–queues, maps, and vectors.

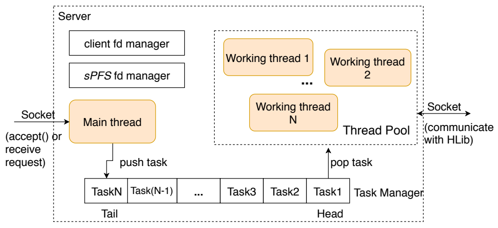

Fig. 3. Structure of SPS. The rounded rectangles represent threads

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 an STL *Map* acting as a key-value table where the key is a physical path and the value is a data structure that describes an entry of a logical file according to its declaration in different $sPFS$. Each time an open request arrives, the responsible working thread will acquire the corresponding *fd* from the fd manager if it exists, otherwise, a new fd will be created and inserted into the manager. The existence of a *fd* indicates that an $sPFS$ file is opened and is ready for writing data.

2) HLib: Hooking Library: The Hooking Library(HLib) is a dynamic library linked to applications. The HLib keeps LDPLFS's fundamental functionality that intercepts POSIX operations [19] and redirects them to another field while modifying the destination of write requests from a local write request to a server process. HLib also modifies the $write()$, $open()$, and $close()$ functions to deal with the write synchronization issue when a FUSE is absent. Without addressing the synchronization, concurrent write operations to a single file will turn into writes to separate files with the same name. These misaligned writes will cause potential data imbrication and lead to severe data inconsistencies.

We borrow the $-wrap$ functionality with POSIX syntax such as $open, write, read$, and $close$ to hook I/O operations, and use ”*__real_*” prefix to distinguish the original POSIX functions. A process will choose the customized procedure if an I/O request is accessing any of $sPFS$ mount points. Otherwise, the process forwards the request to the kernel. We compile HLib as a dynamic library with an extension ”*.so*” and link the library to applications with Linux *loader* during the execution. We utilize a tmp file to represent an $sPFS$ file so that the stackable file system layer is transparent to applications. An application triggers I/O operations with the help of the corresponding tmp file's fd. HLib then maps the tmp file's fd to a $sPFS$ file fd and interprets the triggered I/O operations to $sPFS$ forms. Besides, HLib is responsible for communications between the application and the server process. This responsibility requires HLib to maintain a socket for all the client processes.

3) CStore: Customized IOStore: IOStore refers to those handles of $sPFS$ utilizing to process I/O operations with interfaces of underlying file system such as POSIX and PVFS. CStore works in the server process and requires to modify $sPFS$ IOStore to support diverse underlying storage systems. CStore manages the interfaces as different IOStore objects. When $sPFS$ are writing data or indexing droppings in backends, it used to invoke a corresponding IOStore object to perform write operations on the target file to the chosen backend. Instead of triggering actual data writing, Our CStore is trying to modify the $sPFS$ IOStore to invoke socket communications. Section III-B1 explains the detailed reasons for this approach.

In order to maintain the compatibility, CStore spends major efforts on modifying the $write()$ function in the $sPFS$ IOStore. The modified $write()$ only urge the server process to manipulates the updates of metadata while leaving practical data writing in original client processes.

When a writing request arrives, CStore obtains a socket fd of a client for the subsequent communications from parameters. A message containing the address where the data should be written will be sent to the client process along with the obtained socket fd. The actual data writing is raised by the corresponding client process afterward.

B. 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 applications ones.

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. *Figure 4* 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 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 current *pid*, data count, and physical target path is sent to the server process. Meanwhile, the corresponding process is blocked by $recv()$ so that the customized write procedure does not occupy duplicated CPU thread resources. Note that the message does not contain any effective data at this stage.

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.

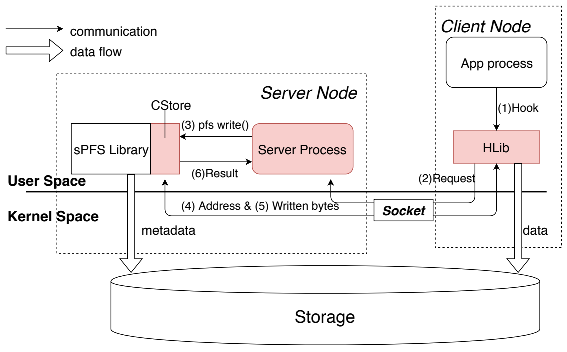

Fig. 4. The procedure of writing a $sPFS$ file

The working thread calls $sPFS$'s $write()$ after verifying a writing flag included in parameters. In particular, the respective client socket fd is deliver 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.

2) Reads: The read operation is simpler than writes in SHC. *Figure 5* presents the data flow of communication and read data streams. The procedure of reading an $sPFS$ file is similar to a common linking library in some way.

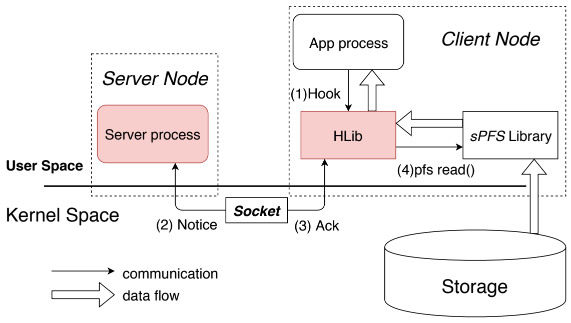

Fig. 5. The procedure of reading a $sPFS$ file

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 storages. 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 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.

C. 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.
- Consider that concurrent I/O operations have large size data buffers, the $read()$ and $write()$ in HLib do not in charge of closing the socket of a client. Only $close()$ deals with the termination of a client socket.
- 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 rooms 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.

IV. Experiment Results and Evaluation

A. Implementation of SHC on PLFS

Parallel Log-structured File System(PLFS) is a typical stackable parallel file system mainly designed for quick checkpoint I/O by the Los Alamos National Laboratory(LANL) and EMC Corporation. Its most brilliant characteristic is to change N-1 I/O pattern into N-N I/O pattern by arranging a container that logically stands for a common file. Multi processes can simultaneously write to this "file" by respectively writing different physical data dropping files in the container, while its metadata is managed by a designed mechanism with the help of some index dropping files [5].

As a user-level file system, PLFS cannot be deployed in an operating system without manipulations. One of the solutions is to use MPI-IO API reconfigure applications' code which is costly since POSIX is widely supported by applications. The other one is to mount PLFS with FUSE. Serving as a transparent middle-ware I/O layer, PLFS draws supports from underlying file systems by managing their interfaces such as PVFS, POSIX, HDFS, etc., in its *IOStores* module. We modify PVFS and POSIX IOStore in our study.

B. Testbed

We implement our SHC on a Sunway TaihuLight HPC testbed cluster to test the I/O concurrency and up-scalability. The cluster for our test is built on an eight-node Lustre file system and is interconnected with InfiniBand. The file system provides 155TB storage capacity.

We deploy PLFS (v2.5) with FUSE (v2.9.7) and SHC, and compare the performance obtained by fs_test benchmark [21]. fs_test is an open source I/O pattern emulation benchmark application developed at LANL. This benchmark can be used to emulate a real applications I/O pattern. It supports the MPI/IO, POSIX, and PLFS API I/O interfaces. For the purposes of this evaluation, 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 OpenMPI (v3.1.0) to manage multiple processes in our tests.

C. Results Analysis

1) Bandwidth: Figure 6 shows the fs_test results of writing and reading bandwidth under different conditions which are file size, transfer size and number of processes. We do not prohibit cache in the reading of both SHC-PLFS and FUSE-PLFS because FUSE's double caching is also an important impact of their comparison. In *Figure 6*, SHC-PLFS shows outstanding performance compared with 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 blocks(nobj). Since SHC has to maintain communications between applications and SPS through sockets for each I/O block, 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 of transfer sizes for 2 file sizes. *Figure 6(a)(d)* and show sharply increasing the bandwidth of SHC with the growth of transfer size. We notice that SHC-PLFS system presents a great superiority compared with FUSE-PLFS. One of the main reasons is that FUSE-PLFS produces more than one memory copying of data, while SHC does not introduce extra data copying between kernel and userspace. When transfer size is 4KB, writing bandwidth of SHC-PLFS is smaller than FUSE-PLFS because the proportion of extra overheads caused by sockets and mutex competition in each operation is larger than the additional memory data copying in FUSE-PLFS. FUSE-PLFS presents low stable bandwidth which can probably blame MPI. When writing to FUSE-PLFS, it is an N-1 pattern in MPI's view, while it is an N-N pattern when

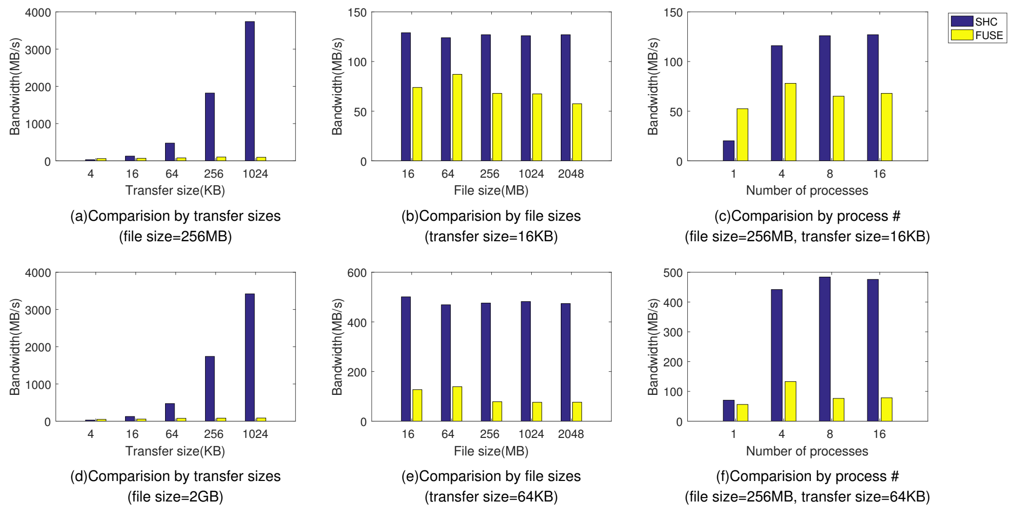

Fig. 6. Comparisons on write bandwidth. (a) and (d) shows the important effect of transfer size in two types of file sizes; (b) and (e) support an analysis that file size may take little influence on write bandwidth; in (c) and (f), SHC shows lower performance in single process case because of the gap between two write operations.

writing to SHC-PLFS. So that the former's synchronization time is much higher than the latter's(Table *I*).

TABLE I
WRITE SYNC TIME OF SHC-PLFS VS. FUSE-PLFS (IN SECOND)

	file size	*transfer size*					
		4KB	*16KB*	*64KB*	*256KB*	*1MB*	
SHC	256MB	2.64	3	2.44	3.26	4.99	$*10^{-3}$
	2GB	0.019	1.73	2.37	3.67	3.49	
FUSE	256MB	0.138	0.893	1.88	1.37	1.6	
	2GB	0.113	1.53	6.31	1.54	3.39	

File sizes: Notice that SHC-PLFS exceeds FUSE-PLFS in *Figure 6(a)(c)*, thus we choose 16KB and 64KB as fixed transfer sizes in the following tests. In *Figure 6(b)(e)* , both SHC-PLFS and FUSE-PLFS present a stable level of bandwidth. Different file sizes effect little on the bandwidth performance caused proportion of effective I/O is determined for each fixed-size transferred block in some way no matter how many 'nobj' a test contains. Comparison of (a) and (d) in *Figure 6* proves proves this analysis either.

Processes Quantity: In *Figure 6(c)* , performances of SHC-PLFS with a small number of processes are much lower than that with more processes and FUSE-PLFS cases. That's because SPS introduce extra waiting time between two writing operations. After a client connects to the SPS through a socket, the main thread of SPS will keep receiving requests from this client and push these requests into the task manager. When a working thread in SPS picks a task from queue, the main thread will stop receiving requests from the exact client so that the following packages from this client have to stay in the socket buffer until the working thread finishes. Additional wait time is introduced to the gap between two requests. In multiple processes cases, the main thread keeps busy receiving requests and the effect of gaps is depressed. In addition, working threads with requests from different clients will compete for locks for synchronization. Although the competitions increase the total time of the operations, more requests from these clients gather in the socket buffer so that the total number of gaps during the whole test decreases. Because more requests will be pushed into the task manager before being picked since the main thread has a higher priority than working threads. When it comes to larger transfer size(*Figure 6(f)* SHC-PLFS reaches a more outstanding level that even in one process case it exceeds FUSE-PLFS.

Reading bandwidth results shows an overall advantage compared to FUSE-PLFS. Since the reading procedure of SHC can directly obtain data from storage except only once communication to the server node for consistency, it gives higher increment than the writing procedure does. We omit the reading procedure analysis in this paper for it sharing similar reasons with the writing one.

2) Open time and close time: Open time of the two system is shown by *Figure 7*. SHC keeps staying at a stable low level of open performance, while FUSE's open time oppositely presents much higher. The open time of FUSE-PLFS shows 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 [2]. There may be numbers of lock competitions among queues in FUSE'e kernel module. In contrast, with SHC, an application can directly invoke $plfs\_open()$ for a file except only one synchronization occurs in SPS for each open operation.

Besides, SHC behaves outstanding when closing a

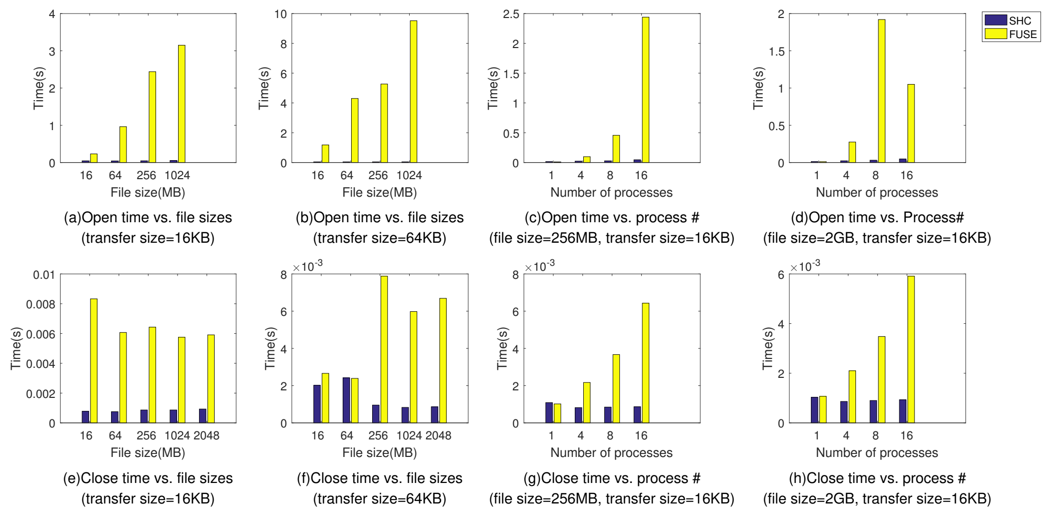
Fig. 7. (a)-(d) are comparision of open time between SHC-PLFS and FUSE-PLFS; (e)-(h) are related to close time. All of the results shows high overheads of FUSE-PLFS because of its internal file handles and synchronization

file(*Figure 7*). Its close time also stays at a stable low level while FUSE's is much higher than it. One reason is that, FUSE will invoke its $FLUSH$ operation [2] which is meaningless for PLFS since $plfs\_close()$ contains index flushing. What's more, FUSE has to maintain its handles of open files in the FUSE kernel module, and in the N-1 pattern, all processes have to compete for locks to update the handle. In contrast, PLFS has such a mechanism similar to these handles, for example, maintaining the reference number of a file. Thus, calling $plfs\_close()$ directly by SHC is relatively more efficient than invoking $close()$ through FUSE. Similarly, multi processes also introduce lock competitions in the FUSE kernel module while closing a file. *Figure 7(g)(h)* reveal negative influences caused by multi-processes.

D. Comparison with LDPLFS

We run additional tests on a 9-node cluster to compare the performance with LDPLFS. The cluster is equpped with Intel Xeon E5-2603 and 16GB DDR4 memory for each node, and runs on PVFS file system. Figure 8 compares performance between LDPLFS and SHC. We can observe that SHC retains similar read performance compare to LDPLFS. But SHC can only reach 65-96% write performance of LDPLFS, which is mainly due to the SHC additional metadata management.

In general, we can argue that LDPLFS *hooking* method works well for data reading and writing. However, when it comes to concurrent writes to a single namespace, it is another story. First of all, if more than one applications try to write to the same file (e.g. IoT sensor network), multiple writes may not be issued on the correct locations and may introduce data overlapping. Such communications amongst the applications can be achieved via FUSE daemons, this mechanism is eliminated when FUSE is bypassed. This could

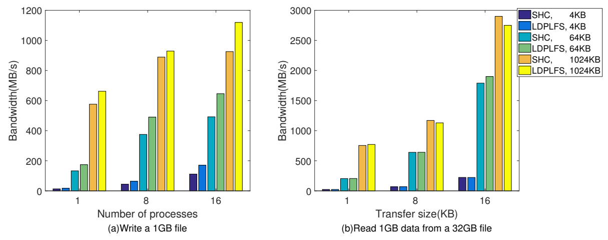
Fig. 8. Both writing and reading comparison are under different number of processes and transfer sizes. (a) Each data block is $sync$ after $write()$; (b) We release cache between two workloads.

lead to a severe data inconsistency problem. Furthermore, LDPLFS does not take care of the mis-calling $close()$ 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.

We present a small experiment to study the potential deficiency of LDPLFS. 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 II 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 stores 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.

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,

TABLE II
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

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 intervene, kernels may not be notified to flush the metadata only if applications raise explicit $close()$ calls.

V. Conclusion

In this paper, we present a mechanism called SHC to reduce kernel crossings of a parallel file system in userspace based on the discussion of the limitation of FUSE when mounting stackable parallel file systems. The SHC is comprising of three components–SPS, HLib, and CStore that utilizes sockets. The SHC is then applied on PLFS and is further implemented on an 8-node Sunway TaihuLight HPC testbed Lustre file system with InfiniBand connections. The preliminary results are collected from the fs_test benchmarks. The results indicate that kernel crossings become a major I/O performance bottleneck in a concurrent I/O circumstance for stackable parallel file systems. The trend that is presented in the results also demonstrates that the I/O performance will be heavily affected by kernel crossings as storage system go upscaling. In addition, we compare the performance between SHC with LDPLFS and reveal its potential to write inconsistency issue under IoT scenarios, where multiple processes write to the same file with communicating with each other.

SHC is eager for tests on petascale clusters to reveal its bottlenecks. Furthermore, we plan to improve the fault tolerant capability for SHC. Moreover, it is further work to make the SHC method general support for more stackable parallel file systems by a group of universal interfaces.

VI. Acknowledgement

Shu Yin's research is supported by the National Natural Science Foundation of China under Grant 61402158, the China Postdoctoral Science Foundation under Grant 2015M572708, and ShanghaiTech University under a startup grant. Xiaomin Zhus 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] Wikipedia. (2018, May) Filesystem in userspace. [Online]. Available: https://en.wikipedia.org/wiki/Filesystem_in_Userspace

[2] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of user-space file systems," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 59–72. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

[3] K. Ren and G. Gibson, "TABLEFS: Enhancing metadata efficiency in the local file system," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 145–156. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren

[4] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems," in *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, 2015.

[5] 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 *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[6] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra incognita: On the practicality of user-space file systems," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. Santa Clara, CA: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/tarasov

[7] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 206–213. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774130

[8] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra incognita: On the practicality of user-space file systems," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. Santa Clara, CA: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/tarasov

[9] R. H. Storage. (2011, June) Linus torvalds doesn't understand user-space file systems. [Online]. Available: https://redhatstorage.redhat.com/2011/06/28/linus-torvalds-doesnt-understand-user-space-storage/

[10] ceph.com. (2011, July) Linus vs fuse. [Online]. Available: https://ceph.com/dev-notes/linus-vs-fuse/

[11] K. Munir, M. S. Al-Mutairi, and L. A. Mohammed, *Handbook of Research on Security Consideration in Cloud Computing*. IGI Global, 2015, ch. Chapter 17:Achieving Efficient Purging in Transparent per-file Secure Wiping Extensions.

[12] E. Zadok and I. Badulescu, "A stackable file system interface for linux," 04 1999.

[13] E. Zadok, I. Badulescu, and A. Shender, "Cryptfs: A stackable vnode level encryption file system," 06 1999.

[14] K. S. Yim and J. C. Son, "SynergyFS : A Stackable File System Creating Synergies between Heterogeneous Storage Devices," *Performance Evaluation*, 2008.

[15] J. Sipek, Y. Pericleous, and E. Zadok, "Kernel support for stackable file systems," in *In Proc. of the 2007 Ottawa Linux Symposium*, 2007, pp. 223–227.

[16] B. Cornell, P. A. Dinda, and F. E. Bustamante, "Wayback: A user-level versioning file system for linux," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 27–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247415.1247442

[17] Y. Zhou, Y. Deng, L. T. Yang, R. Yang, and L. Si, "Ldfs: A low latency in-line data deduplication file system," *IEEE Access*, vol. 6, pp. 15 743–15 753, 2018.

[18] J. Shu, Z. Shen, and W. Xue, "Shield: A stackable secure storage system for file sharing in public storage," *Journal of Parallel and Distributed Computing*, vol. 74, no. 9, pp. 2872 – 2883, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001051

[19] S. A. Wright, S. D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S. A. Jarvis, "Ldplfs: Improving i/o performance without application modification," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 1352–1359.

[20] T. Wang, K. Mohror, and A. Moody, "Direct-FUSE : Removing the Middleman for High-Performance FUSE File System Support," 2018.

[21] fs test. (2017) fs-test. [Online]. Available: https://github.com/fs-test/fs_test