Chfs: An Incremental User-Space File System Based on Fuse

Hui Chen
Louisiana State University
Baton Rouge, Louisiana
hchen46@lsu.edu

ABSTRACT

User space file system has been widely used in many production systems because of its ease of development and flexibility. Especially after Fuse was released, there have been numerous user-level file system been developed for various application requirements.

In this paper, we would introduce a user level incremental file system —chfs,which is implemented based Fuse library. Chfs could transparently store the copies of files which under specified size limit. Customers could find out previous versions of the file in the background native file system. Although the read/write bandwidth performance of Chfs is only half of background Ext4 file system, This incremental file system is very useful for the disc storage media like CD/DVD, which could be only written once.

General Terms

Computer System

Keywords

Incremental File system, Performance, Fuse

1. INTRODUCTION

It's a challenge task to develop a in-kernel file system for Unix even it has the module interface support to facilitate the integration of in-kernel file system. Because developers need to understand and familiar with complicated kernel code and data structures, it's prone to generate bugs caused by programming errors, also it's hard to debug the kernel code if the bug crashed down the whole system. Furthermore, there is a steep learning curve for doing kernel development, which has only one programming language choice —C, also the normal C standard library is not usable in the kernel which was not allowed to link with user-level libraries. All these factors stimulate the efforts for developing file systems in user space.

Different from kernel development, programming in user space mitigates several of the above-mentioned issues. Developers could use different programming languages to quickly extend the features of user-level file system. As an example, the popular used HDFS file system was developed in Java with special support to large volume data processing in Hadoop platform. Meanwhile it provides good performance in scalability and reliability in the large scaled distributed computing environment. The abundant programming framework, third-party tools and libraries make it efficient to implement a user-level file system, also the debug for user-level file system is as same as the normal application development, which relief the nightmare sometimes happened in in-kernel file system development.

FUSE [2] is a widely used framework for user-space file system development. It allows non-privileged users to implement any kind of functions based on the interface provided by fuse library. It has been merged into the Linux kernel, while ports and language-bindings are available for several mainstream operating system and system programming languages. Usually developers only need implement the traditional file system operation interfaces provided by fuse_ops. You can compress the data before writing to disk, or even you could read data from your cloud storage address. All these features could be easily extended based on Fuse framework.

Our incremental file system Chfs is also implemented based on Fuse. The main function of it is to save one copy of the file when it is to be "overwritten". When an updated file is closed, it will produce a new file with the same name and be hidden in the same directory. In other words, files in this file system are allowed to be written only once. Users are unaware of such internal details and only see one copy of the files. Also users could specify a size threshold for incremental update such as 1KB or 1MB. Obviously, Chfs has more latency to write than traditional file write operation, but the mechanism implemented in Chfs is useful for the only written once storage media such as CD/DVD. We have conducted extensive experiments to compare the performance of Chfs under different size threshold incremental update. Results indicate that Chfs could achieve best performance with size threshold set to 2KB. Also we compared the performance of Chfs to native Ext4 and example Fusexmp file system, which shows that performance of Fuse-based user space file systems have nearly half bandwidth performance compared to native Ext4 file system.

The remainder of the paper is organized as follows. Some background information about Fuse structure will be introduced in Section 2. Then the implementation detail of Chfs is demonstrated in Section 3. Section 4 presents the experiments environments and benchmarks, also demonstrate the experiments results analysis. Section 5 concludes the whole paper.

2. BACKGROUND

Fuse framework composed of three modules: the fuse library (libfuse), the kernel module (fuse.ko) and a mount utility (fusermount). The kernel module responsible for capturing the I/O system calls from user applications, then it will invoke the corresponding user-space file system code to process the system call. In this process, FUSE kernel module and the FUSE library communicate via a special file descriptor which is obtained by opening */dev/fuse*. This file could be opened multiple times for different user-space file systems, the fuse library will match each different file descriptor to different mounted filesystem.

The third component of fuse is *fusermount*, which was invoked every time when we mount or unmount the target point. It's very useful for non-privileged users to mount the file system and specify the parameters at the time of mounting the file system.

The benefits of Fuse is obvious. 1) It make kernel transparent to users and developers. Developers could quickly implement a file system without modifications to kernel code. 2) It is usable by non privileged users. 3) It's easier to debug the file system at user-level than kernel-level, the fault in file system will not crash down the whole system.

On the other hand, the pitfalls of Fuse are also apparent. Because there are more execution phases than traditional file I/O system calls, the latency is evident in our experiments. also when large volume I/O pressure comes, the user level file system will be not so stable as usual.

2.1 How does FUSE Work?

Based on the introduction on fuse, we knew there are actually two parts running for a user-space file system. One part is running in the user space, the other part is running in the kernel space. In order to implement ourself user-space file system based on FUSE, we need to understand how FUSE work in the whole process.

Steps to launch the user-space file system:

- **First Step:** when the user mode program calls fuse_main(), which parses the arguments passed by the user mode program, then calls fuse_mount()
- **Second Step:** fuse_mount creates a UNIX domain socket pair, then forks and execs fusemount with one end of the socket embedded in the FUSE_COMMFD_ENV environment variable.
- **Third Step:** fusermount utility makes sure the fuse kernel module is loaded. Then open /dev/fuse and send the file handle over a UNIX domain socket back to fuse_mount().
- **Fourth Step:** fuse_mount() returns the file descriptor for /dev/fuse to fuse_main().
- **Fifth Step:** fuse_main() calls fuse_new() which allocates the struct fuse data structure that stores and maintains all the metadata information of the mounted file system.
- **Sixth Step:** As the last step, fuse_main() calls either fuse_loop() or fuse_loop_mt() which both start to read the filesystem system calls from the */dev/fuse*, then invoke the user mode functions stored in fuse_operations data structure before calling fuse_main(). The results of those calls are then written back to the */dev/fuse* file where they can be forwarded back to the system calls.

The other part is the kernel part, which is running to capture the system calls related to */dev/fuse*, then dispatch calls to either request_send(), request_send_noreply(), or request_send_noblock(). The latter two methods are both similar in function to request_send() except that one does not need response and the other is non-blocking. These methods will add the request to "list of requests" structure, then waits for a response.

The request processing component is in the proc filesystem(kernel/dev.c), which is monitoring the */dev/fuse*, responds to file I/O requests to this device. The corresponding read/write methods are fuse_dev_read() and fuse_dev_write(). The kernel and user space program exchange request/response through device */dev/fuse* is clear illustrated in these steps.

3. IMPLEMENTATION

Based on those understanding of FUSE framework, we implemented Chfs based on FUSE library. In the beginning, we need to install the latest FUSE source code in our system. The fuse library version used in Chfs is 2.9.4, which is easy to install on different platforms. We have referred to the examples provided in the FUSE source code, such as fusexmp and fusexmp_fh and so on. According to the goal of Chfs, we only need to make following revisions:

- **Arguments Processing:** we need to specify the size threshold value in the command line such as "./chfs /tmp/fuse [1-9][k|m|K|M] ...", the third argument specify the size threshold value. We need to parse it and pass the value to variable **LIMIT**. Also we need prompt error information if the given arguments are not correct.
- **readdir() revision:** The default mount point is "/", so we could find every directory under "/" after mounting. I have changed this method to only show up "/fuse" directory under "/", also no hidden file or directory will be listed when user issue the command "ls".
- **write() revision:** In order to keep the old copy of file, we need to copy the data to another file after the normal write operation finished. The old file will be named with a timestamp tage after the same file name. Also the old file is in hidden state with a "." at the beginning of file name.

After the implementation, we could find the write operation processed as illustrated in Figure 1.

- **STEP 1:** User's application issue a write request to the mounted fuse directory.
- **STEP 2:** FUSE kernel module intercept the system call to that file descriptor, then dispatch the request to */dev/fuse*, then Chfs will read the request in the

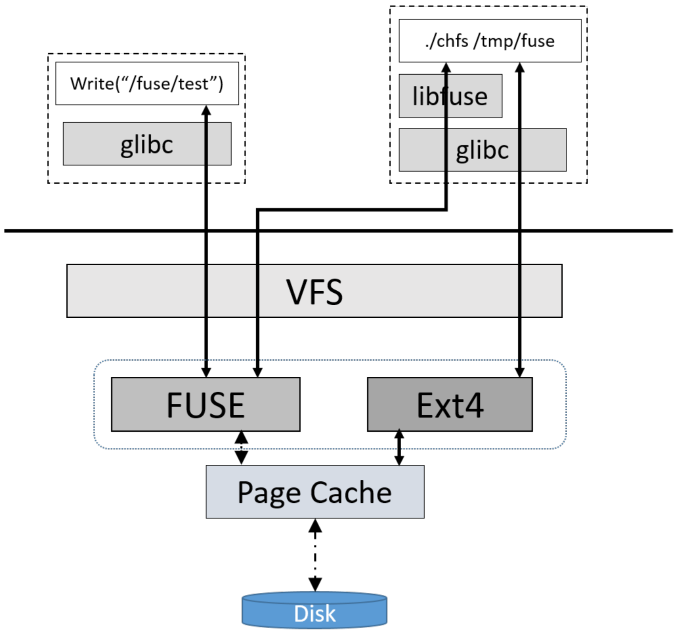

Figure 1: Write System Call Path Through Chfs for a write() operation

fuse_loop(), and invoke the revised write method to finish the request, meanwhile a new copy of the file will be created and stored in native background Ext4 file system through the C standard library methods.

- **STEP 3:** User mode program put response on */dev/fuse*, the fuse kernel module will get the response and forwarded to user application.

Then the whole process finished. For other methods, there is no big changes.

4. PERFORMANCE EVALUATION

Our goal is to find out the optimum size threshold value for Chfs and compare its best performance to fusexmp and native file systems. So we have conducted a extensive experiments on our lab server, which has 6 Xenon E5-2640 running at 2.50GHz , 16 GB of main memory, and a 1 TB LSI MR9240-8i disk. The operating system is Redhat Workstation 6.7 with kernel version 2.6.32, while the version of the user space FUSE library is 2.9.4. The native file system is Ext4.

4.1 Tested Configuration

In this report, we have compared the performance of following file system configurations:

- **Native**- the underlying Ext4 file system.
- **Fusexmp**- the original Fuse file system included as an example in the Fuse-2.9.4 source code.
- **chfs**- the user-level incremental file system developed for this project.

Table 1: Parameters of FileServer

nfiles	10000
meandirwidth	20
meanfilesize	128K
nthreads	50
iosize	1M
meanappendsize	16K

4.2 Benchmark Introduction

The benchmark used in our experiment is Filebench, which is a comprehensive file system benchmark [1] allowed to generate a large variety of workloads. Also it provides interface for user to customize applications' behavior via extensive Workload Model Language (WML). It shipped with more than 40 pre-defined personalities, including micro workloads like sequence read/write, create, delete and writefsync and so on, also complex applications like email, fileserver, webserver,etc.. Moreover, it's easy to set up and quick to get the result.

In our experiment, we chose the fileserver personality, which could emulate simple file-server I/O activity. This workload performs a sequence of creates, deletes, appends, reads, writes and attribute operations on a directory tree. 50 threads are used by default. The workload generated is somewhat similar to SPECsfs. During the experiment, it will generate a fileset with 10000 files, average directory width is 20, average directory depth is 3, total data volume is 1240MB. So this workload is very close the real workloads in the fileserver. Detail information could refer to Table 1.

4.3 Results Analysis

The results for different size threshold value is presented in Figure 10, and comparison of native file system, fusexmp and Chfs is illustrated in Figure 13. Next we will give the detail analysis of these results.

4.3.1 Performance Under Different Size Threshold

In order to test Chfs's performance under different size threshold to update, we have chosen 13 different sizes from 1KB ˜4096KB namely 4MB. There are two metrics we presented in these figures. One is operation latency which means each operation cost in the file system, the other is operation bandwidth which indicates how much data has been read/write during the unit time period in the file system.

For read,write and append operations, we have both metrics presented in the figure. Because create and delete has no data to transfer, both bandwidth metric are 0MB/s. But for the bandwidth performance of read, write and append, we could find nearly the same pattern in these three Figures 357. But the absolute value of read is better than write, which is better than append. Because the average append is only 16KB, which affect the bandwidth of append operation, which could achieve higher bandwidth if the append size is bigger.

For the latency metric, we could find all the five operations have different pattern. Read operation's patter is close to create, the left three —write, append and delete —have more similar pattern. Because the absolute value is very small, the trivial difference is ignorable. Meanwhile, those wave changes do not hinder us to find out the optimal configuration for the size threshold, which should be 2KB to

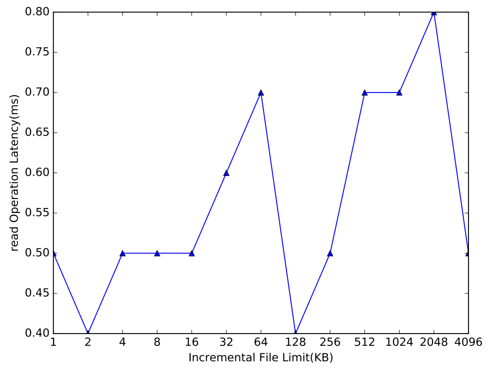

Figure 2: Read Operation Latency

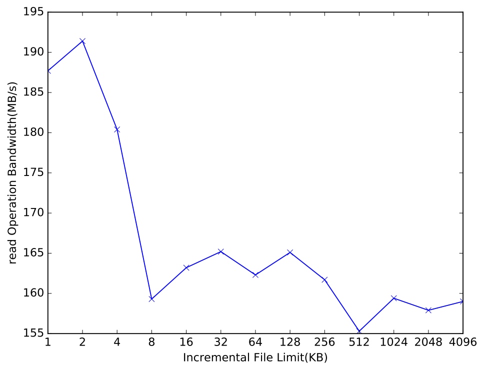

Figure 3: Read Operation Bandwidth

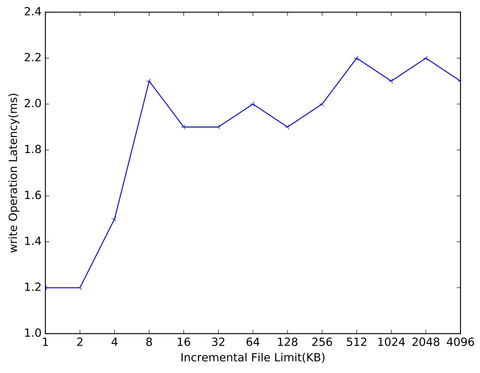

Figure 4: Write Operation Latency

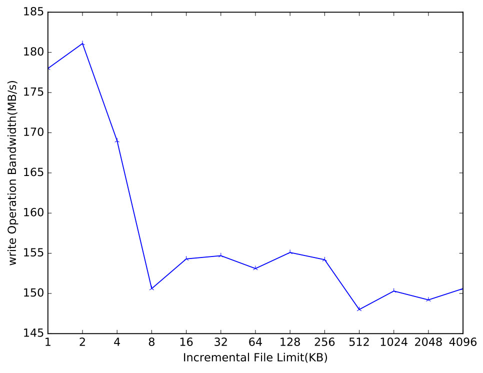

Figure 5: Write Operation Bandwidth

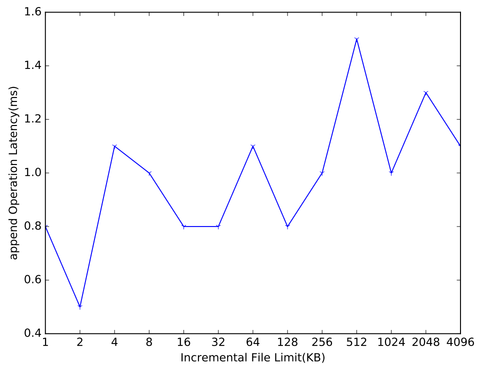

Figure 6: Append Operation Latency

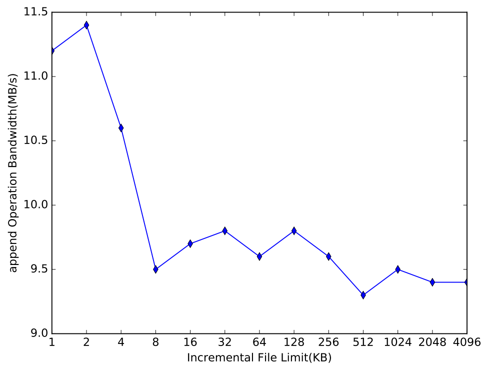

Figure 7: Append Operation Bandwidth

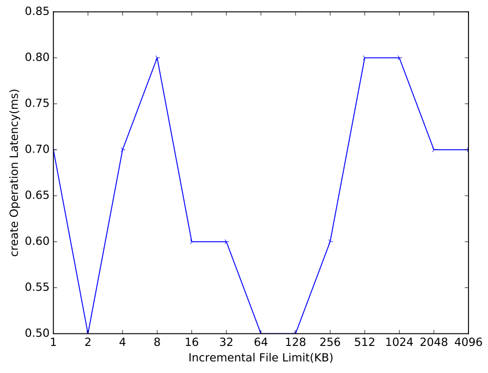

Figure 8: create Operation Latency

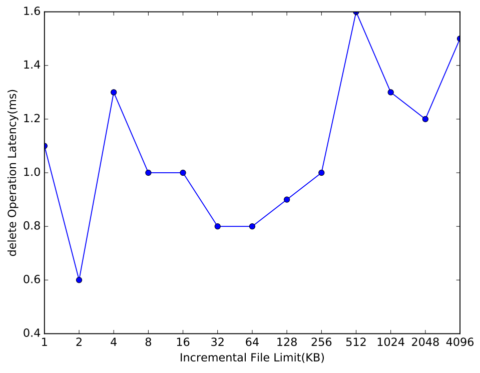

Figure 9: Delete Operation Bandwidth

Figure 10: Performance of Chfs Under Different Threshold Size

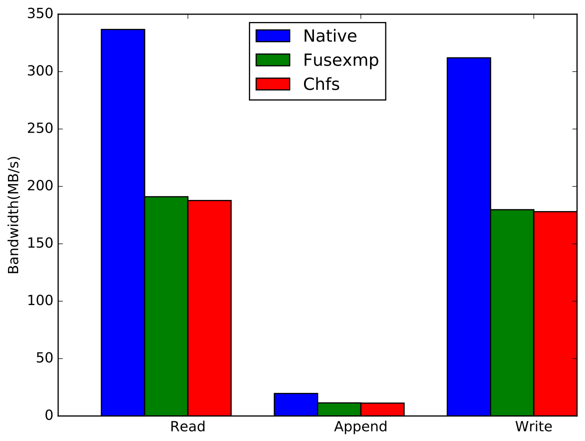

Figure 11: Bandwidth Comparison

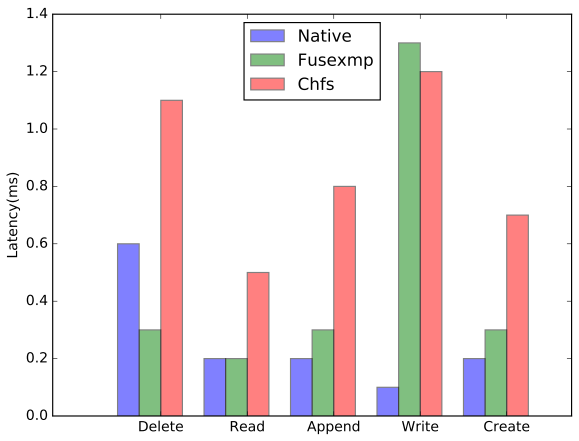

Figure 12: Latency Comparison

Figure 13: Comparison of Native, Fusexmp and Chfs

achieve the best performance for all kind of operations.

4.3.2 Comparison among Different File Systems

We have compared three different file system in the Figure 13. For Chfs, we chose the best performance when the size threshold value set to 2KB. In Figure 11, We could find that the native Ext4 has better (nearly double) bandwidth performance than Fusexmp and Chfs, which proved that overhead in the user space file system is obvious. But between Fusexmp and Chfs, the bandwidth difference is ignorable, because bandwidth only statistic how much data has been transferred. Whereas Chfs has to store another file after write operation, it doesn't affect the bandwidth performance of Chfs.

Figure 12 illustrated the latency of these three file systems for following operations: delete,read,append,write and create. In our expectation, for each operation, native should better than fusexmp, and fusexmp should better than Chfs. We could find that Chfs is worse than native file system for each operation, which is in our expectation for the latency introduced by two round of context switch between user and kernel spaces. But for some operation like delete and read, fusexmp is better than or same as native file system, which beyond our expectation and I do not know how to explain it. Also for write operation, Chfs is better than fusexmp, which is unbelievable even though there is only 0.1ms difference, because Chfs have two write operations in each write call.

5. CONCLUSION

In this project, I have learned how FUSE to support the user space file system, and implemented a user space filesystem Chfs which could do incremental update to files smaller than specified size threshold. Through extensive experiments, I have experienced how 'Bad' Chfs is, which is in my expectation. Anyway, I still look forward to build something 'better' based on Fuse in the future.

6. ACKNOWLEDGMENTS

I need to appreciate Prof. Chen's help for this project. I even do not know the existence of FUSE without his project introduction.Although the filesystem implemented in this project is not so useful, I still learned a lot during this project.

7. REFERENCES

[1] R. McDougall and J. Mauro. Filebench tutorial. *Sun Microsystems*, 2004.

[2] M.Szeredi. Filesystem in userspace. http://fuse.sourceforge.net/.