M.S. THESIS

Eliminating FUSE Bottleneck by Implementing Multi-threaded Framework on Distributed File System

분산 파일 시스템의 퓨즈 멀티 쓰레드 구현을 통한 퓨즈 병목 현상 제거

AUGUST 2017

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Sophal HONG

M.S. THESIS

Eliminating FUSE Bottleneck by Implementing Multi-threaded Framework on Distributed File System

분산 파일 시스템의 퓨즈 멀티 쓰레드 구현을 통한 퓨즈 병목 현상 제거

AUGUST 2017

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Sophal HONG

Eliminating FUSE Bottleneck by Implementing Multi-threaded Framework on Distributed File System

분산 파일 시스템의 퓨즈 멀티 쓰레드 구현을 통한 퓨즈 병목 현상 제거

지도교수 염헌영

이 논문을 공학석사학위논문으로 제출함

2017 년 06 월

서울대학교 대학원

컴퓨터 공학부

홍쏘펄

홍쏘펄의 석사학위논문을 인준함

2017 년 07 월

위 원 장	장병탁	(인)
부위원장	염헌영	(인)
위 원	엄현상	(인)

Abstract

Eliminating FUSE Bottleneck by Implementing Multi-threaded Framework on Distributed File System

Sophal HONG
School of Computer Science & Engineering
Collage of Engineering
The Graduate School
Seoul National University

A popular Filesystem in Userspace (FUSE) is widely used as a data accessing protocol on many modern distributed file systems. FUSE works as a bridge for transferring requests from user application to FUSE-based file system. Actually, it receives requests from user application and then processes some corresponding FUSE operations sequentially by a single FUSE thread. Unfortunately, when a FUSE-based distributed file system performs on workload that consists of a relatively huge number of small-file operations or a very large number of clients, it suffers from overhead of request handling caused by the single FUSE threat. This overhead is the bottleneck of the whole system which significantly degrade the overall performance. In this paper, we propose a multi-threaded FUSE framework that receives and processes requests in parallel which can eliminate the bottleneck caused by that original single FUSE thread. As long as there are requests still available waiting inside the FUSE queue, the other new FUSE thread will be automatically created to receive request and performs some specific FUSE operations simultaneously. We incorporated our mechanism

into a GlusterFS distributed file system. The experiment results of our proposed mechanism indicated that depending on the workloads and hardware used, the performance upgradation is improved by 32% on small-file writing workload and 35% on small-file reading workload.

Keywords: FUSE bottleneck, Multi-threaded FUSE, GlusterFS

Student Number: 2015-23300

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed file system is a widely used file system which is built as client/server based application that can perform I/O operations very well with large amount of data without losing benefits of high scalability, reliability, and performance. In distributed file system, server node performs as a file storage for storing distributed data, where multiple clients can access this data as it were stored on their own resources. GlusterFS, one of the most popular distributed file system, is a highly scalable file system which can support up to several thousands of clients and several petabytes of data [8].

In GlusterFS, data can be stored in many different ways based on volume type to ensure scalability, reliability as well as performance. Moreover, there are a lot of useful features provided on configuration setting to leverage performance and other benefits based on workload and hardware used. GlusterFS supports several data accessing protocols to allow clients to access data stored on server nodes. One of the most popular data accessing protocol is FUSE (Filesystem in Userspace) [20], which is used as a Gluster Native Client accessing protocol. We are going to discuss more about FUSE in the next section.

GlusterFS is implemented as a multi-threaded application that contains

several threads running concurrently to perform their own specific jobs. When a program is implemented as a multi-threaded application, there must be some possible problems that we should take care of like concurrency, consistency, as well as bottleneck caused by any thread. Unlike concurrency or consistency problem, the bottleneck problem is somehow not a fatal error which causes the whole system to be crashed or produces unexpected result. Even though the system suffers from bottleneck problem, it still lets the whole system to be correctly deterministic but performance is significantly decreased.

When GlusterFS uses FUSE-based native client protocol to access data, there will exist one FUSE thread works as a bridge to receive request from user application, performs some FUSE operations and pass on request to I/O threads to perform specific I/O operations. Multiple user applications can make requests as many as needed; I/O worker application can also create new I/O thread as many as the number of requests waiting for processing, while there is only one single FUSE thread is used in GlusterFS to receive requests. Therefore, when a large number of requests is performed at the same time, the single FUSE thread is unable to receive those requests quickly, and perform as fast as the multiple I/O threads do. This is the bottleneck caused by single FUSE thread which degrades the overall performance for the whole system.

In this paper, we proposed a mechanism to implement a multi-threaded FUSE framework to eliminate the bottleneck caused by the single FUSE thread. Every FUSE thread can receive and process one request at a time simultaneously. After each thread received the request and before processing FUSE operations, they will first check whether there are more requests still available waiting in queue and none of FUSE threads is available to serve request. If so, it will automatically create new thread to handle the request. Our implementation ensures that multi-threaded FUSE framework implemented in GlusterFS can increase its work performance better than the original single thread when performing on workload that consists a large number of small-file requests. Our

experiment results also indicate that at least 35% is improved for small-file reading workload, and 32% is improved for small-file writing workload. This performance upgradation depends on workload and hardware used, which we are going to explain more in discuss section.

Chapter 2

Background

Since the world of computer science and technology is rapidly improving and revolutionizing in which data information is unpredictably growing, we need the data that can be shared between users in order to reduce the amount of data used by each user. Therefore, in order to deal with this requirement, we need to build a system that we can access data efficiently without any problem whether the data we stored is structured or unstructured. As the result, the precious idea of developing a powerful distributed system has been innovated and got more popular than traditional isolated standalone system. Significantly, a distributed system is a combination of several independent computers connected to each other that appears to users as a single coherent system [19]. Distributed computing systems provide a huge range of advantages over standalone computing systems where its data is stored in a distributed way with several connected nodes as servers.

2.1 Distributed File System

Distributed file system is a widely used file system which is built as client/server-based application. Server works as a centralized storage system that provides common sharable data where clients can access and process this data as it were located on their own storage device. In distributed file system, data is stored on server computers and can be distributed efficiently to a group of connected client computers. So that, client computers can access data easily without using their own resources. Nowadays, while data-intensive science is increasing, highly scalable distributed file systems have played an important role to ensure that data is secured, ease of uses, sharable, and its performance is acceptable.

The advantages of distributed file system are (1) it can be easily mounted and manipulated by every library or program whether it is written in the last long years ago. That means we do not have to convert those aged programs to recent version in order to use their features or data; (2) it can be efficiently shared between multiple systems. Everyone can access to mounted file system simultaneously and process data stored in that centralized storage as needed; (3) it can be highly scalable to several hundreds or thousands of computing machines which are connected to each other through network and can perform their particular work at the same time. Due to these advantages, a lot of people changed to use distributed file system instead of traditional local file system. As the requirement, many researchers have been working on file system technology and have made several beneficial distributed file systems [1 – 18].

2.2 GlusterFS

GlusterFS is an open source application developed as a distributed file system which can support several thousands of clients and enable them to store multiple petabytes of data. It is one of a popular file system built with a stackable design, modular and especially no-metadata server architecture. When there is

no metadata is stored on server, it obviously ensures that GlusterFS can provide higher performance as well as linear scalability and reliability. Moreover, in order to get better performance and highly available enterprise storage over the traditional file system, GlusterFS can also be flexibly combined with commodity physical, virtual, and cloud resources [8].

The GlusterFS has become more popular based on several advantages: (1) Performance - metadata is eliminated from server which can dramatically improve performance and easily to tune performance features to the corresponding storage environment; (2) Elasticity - size of data can be flexibly increased and decreased; (3) Linear scalability - several petabytes and beyond can be supported and thousands of clients can be connected to access data; (4) Simplicity - it is built to run as user space file system which is easy to manipulate and be apart from kernel; (5) Open Source - it is free to download both application and source code which supported and maintained by Red Had Inc, as part of Red Hat Storage.

The concept of "translators", stackable modules which perform their specific purpose, is used in GlusterFS to allow multiple threads working on their own specific jobs concurrently. In hierarchy structure, translators are stacked on each other in which each performs specific operations by receiving data from previous stack called parent translator, and then passes data down to next stack layer called child translator. In GlusterFS, many various kind of file system attributes can be used and easily to adjust performance features like write-behind, read-ahead, disk caching, load-balancing, self-healing, and etc.

Furthermore, GlusterFS can let you create any kind of file system you prefer based on various volume types. Volume is a logical collection of bricks, a directory or a basic unit of storage, which works as a centralize storage component. Base on storage environment, various types of volumes can be created such as distributed, replicated, striped, distributed striped, distributed replicated, distributed striped replicated, striped replicated, dispersed, distributed dispersed

volume [8]. Each volume type has its own specific purpose of how data is stored and provides different performance, reliability, scalability, availability, etc.

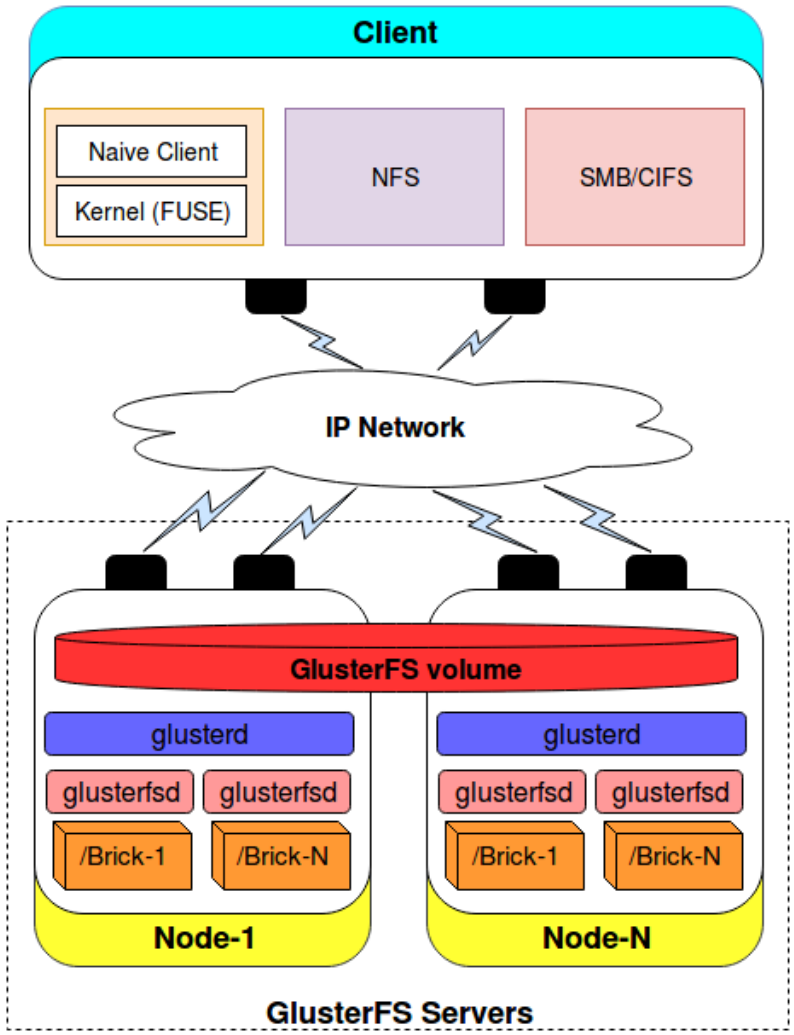

Figure 2.1 Typical GlusterFS architecture

Once the volume is created, we can manage GlusterFS volumes by tuning volume options to improve performance as well as many other beneficial features based on the corresponding hardware environment. In order to access GlusterFS volumes, we need to mount file system on client side to that volume. There are several ways to mount gluster file system and access its volumes: (1) GlusterFS

native client - Filesystem in Userspace (FUSE), which we are going to discuss more in next section; (2) libgfapi - flexible abstracted storage, (3) Network File System (NFS), (3) Server Message Block / Common Internet File System (SMB/CIFS), (4) Gluster for OpenStack - object-based access via OpenStack Swift.

Figure 2.1 indicates a typical architecture of GlusterFS. Clients and servers are clustered together through TCP/IP network connection. A GlusterFS client can use any of data accessing protocols like Gluster Native Client (FUSE), NFS, or SMB/CIFS to mount a GlusterFS volume stored on server. Each node in the GlusterFS server works as a storage pool, which can consist one or more bricks and each brick is managed by glusterfsd daemon. All the server nodes in trusted storage pool requires a glusterd daemon/service to manage and maintain volumes and its cluster membership. In GlusterFS, storage resources from multiple server nodes are combined together and shown as a unified global namespace.

2.3 FUSE

FUSE (Filesystem in Userspace) is a common interface for user-space programs, which provides functionality to export a file system to the Linux kernel [20]. In other words, it is a user-space file system framework which allows non-privileged users to create their own file systems independently without involving any of the kernel code. Three components are needed in order to create and access file system from non-privileged users. One is FUSE kernel module (fuse.ko) which is used as a mediator, on behalf of non-privileged user, to interact with kernel VFS (Virtual File System). Second is a user-space library (libfuse), a collection of file operations like open, close, read, write, etc., which enable programs to access data by using standard file operation system calls. The last one is a mount utility (fusermount), which is used to mount a particular directory to a file system.

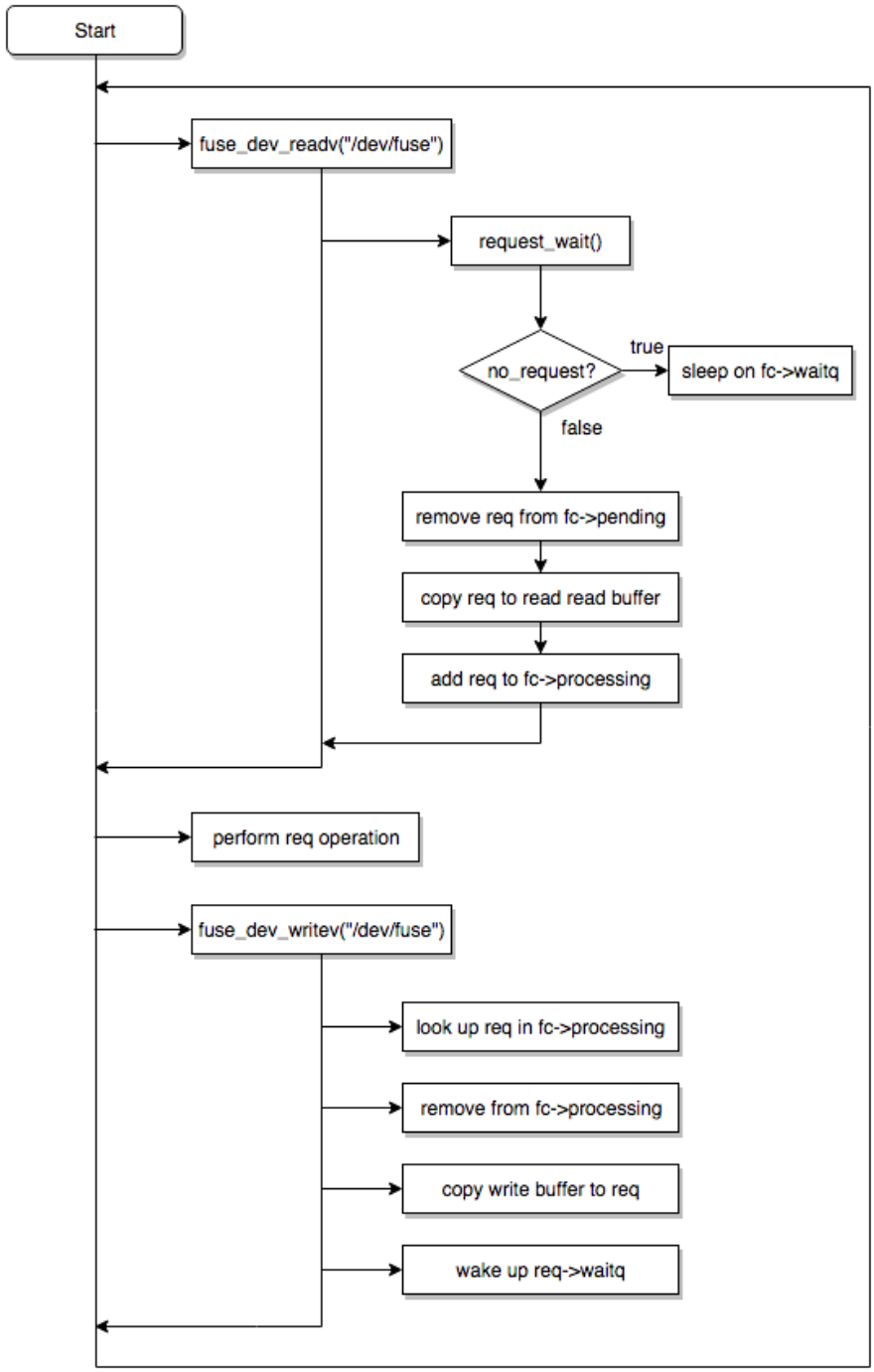

Figure 2.2 The processing flow diagram of how kernel-level FUSE works.

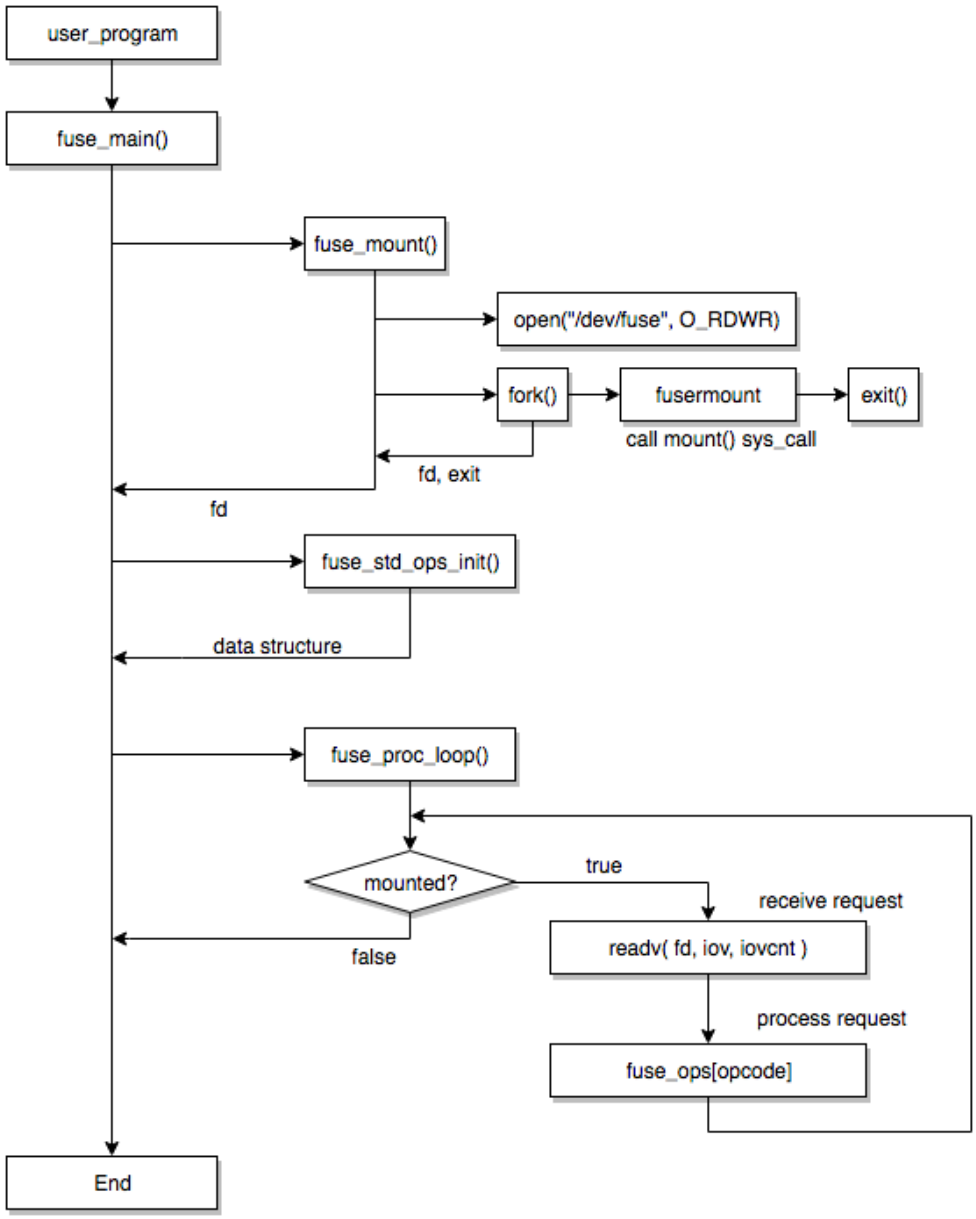

Figure 2.3 The processing flow diagram of how lib-level FUSE user-space works.

Figure 2.2 shows the processing flow diagram of how kernel-level FUSE works. Briefly, in order to handle a complete request kernel-level FUSE needs four steps as follow:

1) FUSE daemon performs *read()* system call on */dev/fuse* and wait for requests at waiting queue (fc->waitq) when there is no request.

2) When user application makes request, it wakes up FUSE daemon to receive request, removes a request to process and waits for result at request waiting queue (req->waitq).

3) Once the FUSE daemon received request, it performs FUSE operations, and return result back to waiting queue then wakes up the requested process.

4) Request process is woken up, then gets result and returns it to the user application.

Figure 2.3 shows the processing flow diagram of how lib-level FUSE user-space works. We are going to analyze how it works on library level in five steps as follow:

1) fuse_main : parses arguments from user process command line like mount point, mount options, etc. then calls function fuse_mount(), fuse_std_ops_int(), fuse_proc_loop() respectively.

2) fuse_mount: calls open() system call to */dev/fuse*, then calls *fork()* to mount file system on *fusermount()* function. Make sure that FUSE module driver is loaded before calling *open()* and *mount()*.

3) fusemount: executes *mount()* system call and sends corresponding file handler back to main process.

4) fuse_std_ops_init: allocates structure and initializes FUSE standard operations.

5) fuse_proc_loop: performs *read()* system call on */dev/fuse* to receive request, and then handles request.

Since FUSE is developed in user-space, it is easy to debug and profiling system while APIs are easily maintainable, and less crash occurrence. That is

to say, user-space code is easy to implement and test unlike kernel code which is complicated and need permission to edit. Even when program is failed or crash during performing, it is easy to clean up and start over. However, when users perform I/O operations on FUSE-based file system, FUSE produces more extra memory copies and context switches for each file system calls [22]. This I/O overhead imposed by FUSE can significantly decrease the overall performance on small-files workloads. Compare to native Ext4 file system, depending on the workload and hardware use, in some cases FUSE can perform as well as Ext4, but in worst cases performance is degraded 3x slower than Ext4 [23].

2.4 Performance Overhead of FUSE

On native file system like ext3 or ext4, there are only two context switches between user mode and kernel mode for each file system operation. One is made by user process making request to kernel virtual file system (VFS) and another one is made by kernel process sending response back to user. However, when FUSE is mounted on file system, it produces two more extra context switches per file system operation. One context switch is caused by user application issues *read()* system call to get request from FUSE kernel module queue, and another one is caused by user application sending response back to FUSE kernel module. Therefore, when accessing data on FUSE-based file system, there are four context switches are needed per each file operation. To give an illustration of how context switch overhead is made by FUSE, let's look at the case of user process make a write request: (1) user application sends request to VFS, then VFS forwards request to FUSE kernel module and put it in queue; (2) FUSE-daemon process gets request by performing read() system call on */dev/fuse* module; (3) FUSE-daemon process processes data and send result back to FUSE kernel module through write system call; (4) FUSE kernel module switches back to user process to deliver result.

According to the previous paper "Performance and Extension of User Space

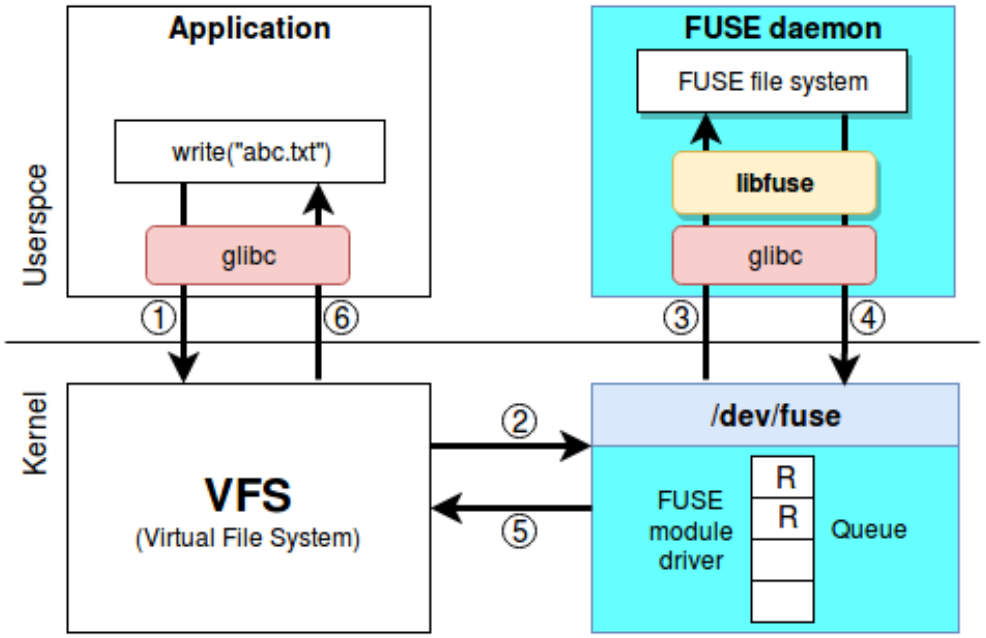

Figure 2.4 A write() system call path through typical FUSE high-level architecture.

File system" written by Aditya Rajagahia and Ashish Gehani, the overhead of FUSE is more visible when the workload consists of a relatively large number of metadata operations, such as those seen in Web servers and other systems that deal with small files and a very large number of clients. FUSE is certainly an adequate solution for personal computers and small-scale servers, especially those that perform large I/O transfers, such as Grid applications and multimedia servers [22].

Figure 2.4 indicates a simplified FUSE high-level architecture of how FUSE handles write request from user application. When a user application makes a request by performing system call to access files located in FUSE-based file system, there are six steps are as follow:

1) user application sends file system call to kernel Virtual File System (VFS).

2) VFS forwards the request operation to FUSE module kernel driver. FUSE

module driver allocates the request structure then put in FUSE queue.

3) FUSE daemon picks up the request from kernel FUSE queue by performing read system call through */dev/fuse*, then processes request by using libfuse library.

4) once the request handling is done, FUSE daemon sends response back to */dev/fuse* by issuing write system call.

5) FUSE module kernel driver marks request as completed.

6) then forwards the result back to user application.

We can see that there are two extra context switches per file operation are added when accessing the file on FUSE-based file system. This extra context switches overhead can be more serious when it is run on a large number of small-file operations workload. In this paper, we are not going to propose mechanism to solve the context switch overhead since it is already solved by using *libgfapi*, an application library which is used to reduce the context switch overhead of FUSE architecture by allowing user applications to directly access GlusterFS volume via native protocol. However, there is another bottleneck problem caused by a single-threaded FUSE-daemon, which we are going to talk more in the next section.

Chapter 3

Problem Statement and Motivation

When the distributed GlusterFS's client uses Gluster Native Client protocol as data accessing protocol to access data that stored in trusted storage pool (Servers), then FUSE-based file system is mounted. GlusterFS application is implemented as a multi-threaded program which contains several threads performing their own specific works. In default, there are at least different kind of eight threads are running in GlusterFS process.

1) *gf_timer_proc thread* – manages and maintains GlusterFS log inject timer event.

2) *glusterfs_sigwaiter thread* – maintains and handles GlusterFS process signal handlers like SIGINT, SIGTERM, SIGHUP, SIGUSR, etc.

3) *syncenv_processor threads* x2 – there can be more than two threads used to check running task count, modify tasks on running queue, waiting queue and swap context, etc.

4) *gf_time_wheel_runner thread* – manages global timer for GlusterFS.

5) *event_dispatch_epoll_worker thread* x2 – by tuning configuration setting, multiple threads can be used to maintain event handlers, socket event handler

for data transportation connection. (default:2, range 1-32)

6) *iot_worker thread(s)* – by default, there are 16 IO-threads which used to perform concurrent I/O operations. These threads are created and dispatched automatically when there are less or more operations in queue.

7) *fuse_thread_proc thread* – a GlusterFS FUSE-daemon thread which is used to receive request from FUSE kernel module driver */dev/fuse*, and then perform FUSE operations for accessing data on Gluster file system.

8) *notify_kernel_loop thread* – use for reverse invalidation of inode.

When we process small-file workload on FUSE-based GlusterFS, the heavily working threads that are always actively running on FUSE-based GlusterFS are *event_dispatch_epoll_worker, fuse_thread_proc*, and *iot_worker* threads. In glusterfs-3-9-0, *event_dispatch_epoll _worker* and *iot_worker* threads are implemented as multi-threaded application, which the number of threads can be flexibly set by easily tuning the configuration setting on volume option properties. The default number of *event_dispatch_epoll_ worker* thread is set to 2, and we can change its number in range of 1 to 32 by performing *#sudo glusterfs volume set test-volume client.event-threads [number].* Then there will exist [number] event threads executing in parallel, which would help process responses faster, depending on the available processing power. Similarly, number of *iot_worker* thread use can be changed by performing *#sudo glusterfs set test-volume performance.io-thread-count [number].* Unlikely, the default number of *iot_worker* thread is set to 16, which means that there are 16 threads are running in parallel to perform concurrent I/O operations. However, it does not mean that all 16 *iot_worker* threads will be created and execute I/O operations; it just the maximum number of *iot_worker* thread which allowed to use. It will be automatically spawn to perform I/O operations only when there are a lot of I/O operations available waiting in the queue, and it will be detached automatically when it sleeps for a while have nothing to perform.

In this paper, what we are going to deal with is the bottleneck caused by

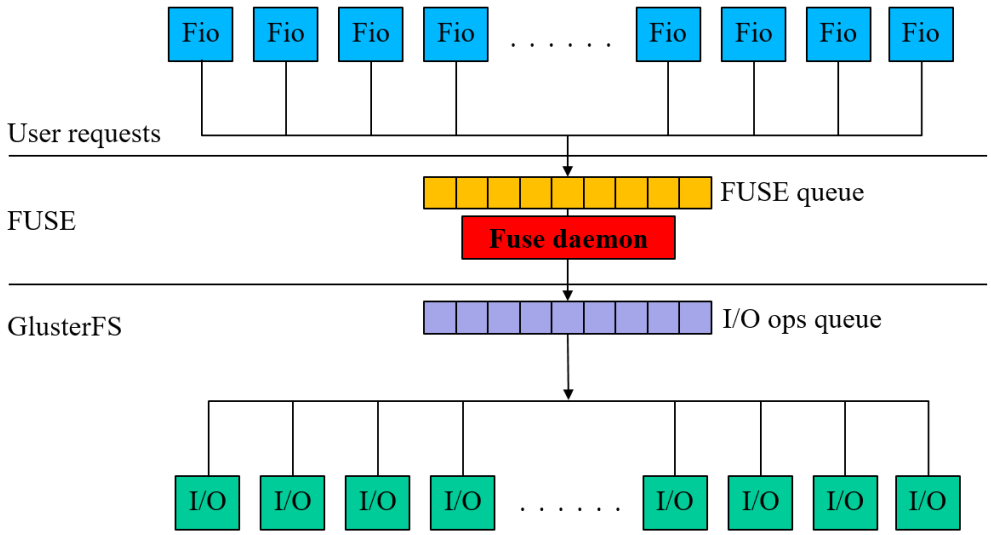

Figure 3.1 The process diagram of how a single FUSE thread receives requests from user application and passes to IO threads on GlusterFS.

fuse_thread_proc thread, which is a fixed single FUSE thread used in GlusterFS process. No matter how much requests are made by users, there will be only one thread is used to receive and handle requests. On the large-file workload, it can perform very well since there are not so many file operations are requested. However, when a large number of relatively small-file operations are requested, the only single *fuse_thread_proc* thread cannot perform as fast as the multiple *iot_worker* threads. To dig a little deeper here, let's see the figure 3.1 of how FUSE thread handles requests when user performs small-file 1KB or 4KB chunk size by executing fio benchmark with 32 threads. The user application makes a lot of small chunk size requests at the same time and the requests are forwarded to FUSE kernel module driver. FUSE module driver allocates request structure and puts them in the pending queue waiting for FUSE-daemon call *read()* system call to take them to process. In GlusterFS, FUSE-daemon is performed by *fuse_thread_proc*, which executes *read* system call on */dev/fuse* to receive request in pending queue. After it received request, it performs some FUSE operations like executing hashing table and resolving file location where

to locate file, etc., then creates frame for request sending, and puts them in queue waiting for I/O operations. Then *iot_worker* threads will get those request frames to perform I/O operations. The single FUSE thread has to execute all those works in sequential mode while multiple I/O worker threads perform in parallel faster than FUSE thread. Since there are a lot of requests are still available waiting in the queue, a single FUSE thread works restlessly and uses CPU up to 100% on core it was running on. This 100% CPU usage shows that FUSE thread is a bottleneck for the whole process, which possibly degrades the overall performance even the I/O worker threads can perform well.

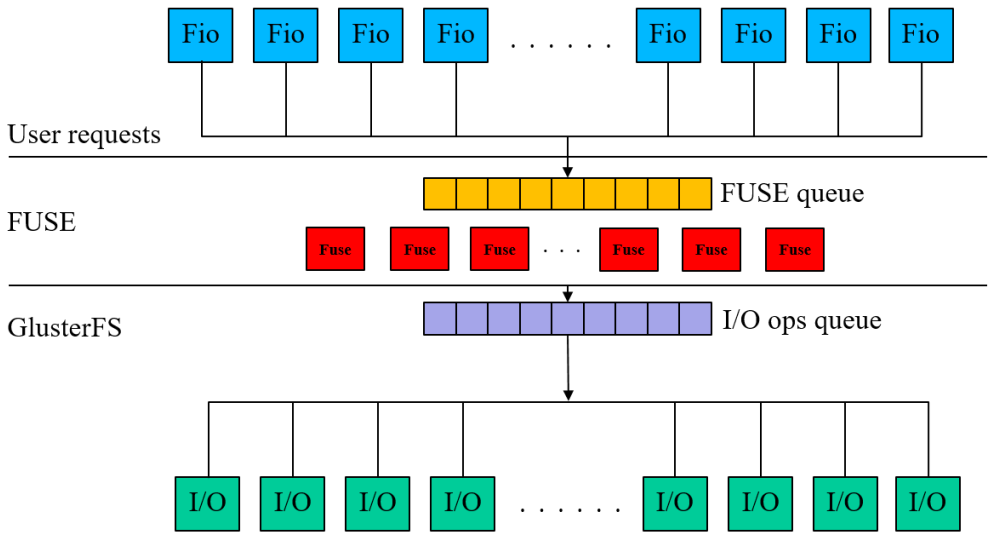

Figure 3.2 The process diagram of using multiple FUSE threads instead of a single FUSE thread on GlusterFS.

To combat this issue, we proposed a mechanism for implementing FUSE multi-threaded application instead of the original single thread running on GlusterFS. Figure 3.2 shows how we implemented multiple FUSE threads instead of a single FUSE thread to receive requests and perform concurrently. Each FUSE thread executes it's works simultaneously in parallel by sharing file descriptor handler of */dev/fuse* module to receive request.

Chapter 4

Design and Implementation

In GlusterFS, requests are made by user applications performing file system calls on client side, where its data is stored on trusted storage pool of server nodes through network connection. To understand more how clients and servers are interconnected to process file system, let's see the picture below.

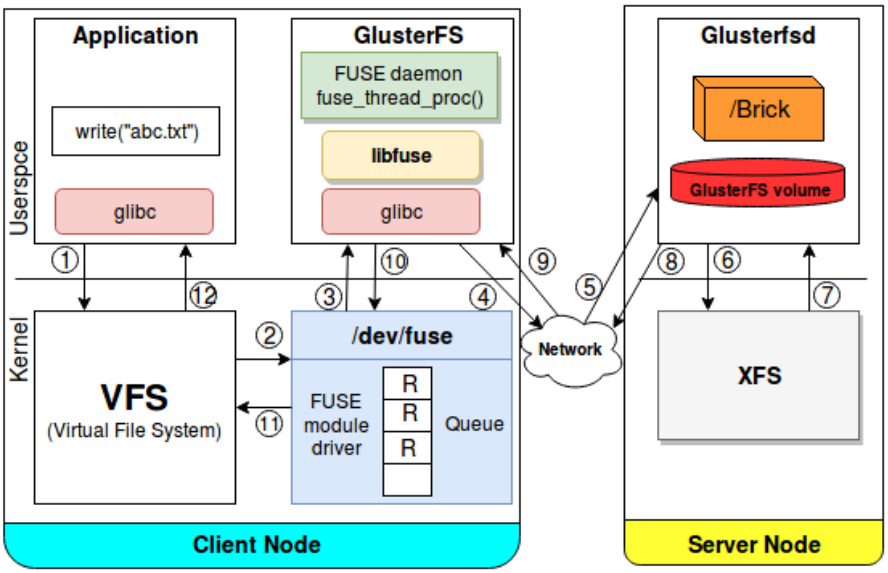

Figure 4.1 A write() system call path on FUSE-based GlusterFS architecture.

Figure 4.1 shows the steps of how GlusterFS performs writing request on FUSE-based file system. On GlusterFS client node, when user application performs write system call on GlusterFS volume mounted as FUSE-based file system, first the request is sent to kernel VFS (Virtual File System) then forwarded to FUSE module driver. FUSE module driver gets the request, allocates request structure, then put in pending queue. GlusterFS's user-space FUSE-daemon (*fuse_thread_proc*) will receive the request by performing *read()* system call on */dev/fuse*, then executes fuse operations to handle request by creating request frame, put in queue, and send to next translator. IO worker threads get request frame, perform I/O operations and send to server node into the corresponding bricks resided on XFS file system. When the I/O operation is done, server node sends response to client node through network again, then FUSE module driver marks request as completed.

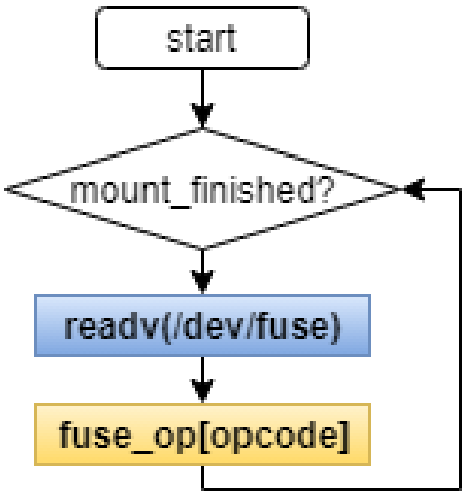

Figure 4.2 The processing flow of the original single FUSE thread.

GlusterFS FUSE daemon is performed by *fuse_thread_proc* thread in GlusterFS process. Figure 4.2 shows the processing flow of how the single FUSE thread performs. To reduce complexity of what FUSE thread executes, we divided works done by FUSE thread as two main functions. One is *readv()* system

call, which is used to receive request from kernel FUSE queue; and the other one is *fuse_ops()*, which is used to perform FUSE operation like what is mentioned above. The original single FUSE thread first needs to check whether or not GlusterFS volume is already mounted. Once the mounting is finished, FUSE thread performs *readv()* system call then executes FUSE operation which corresponding to the request.

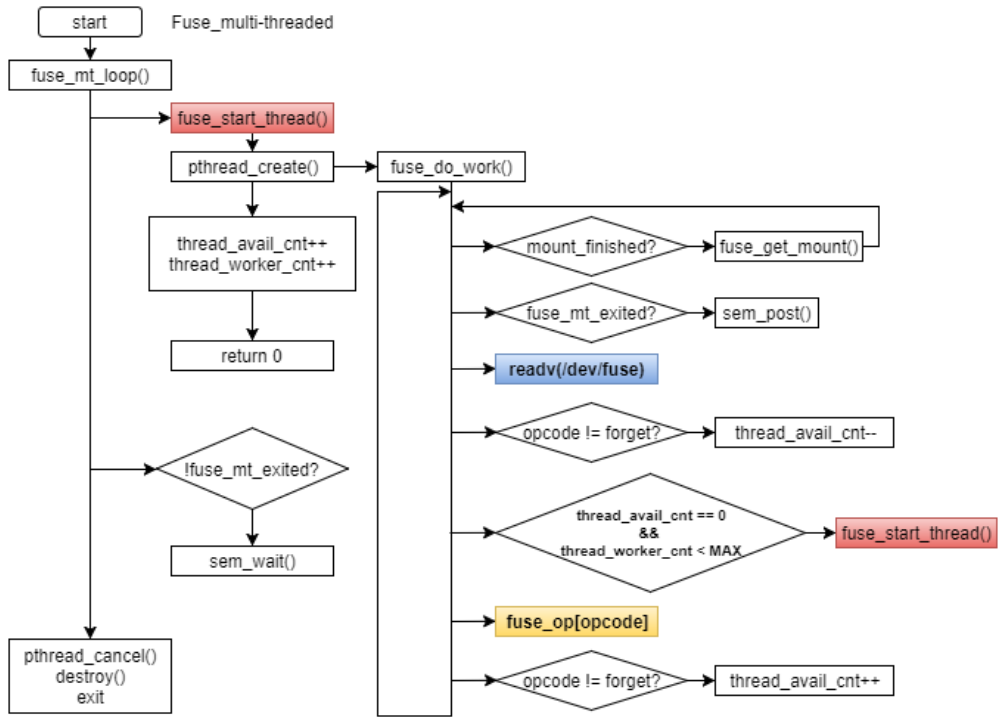

Figure 4.3 The processing flow of multi-threaded FUSE.

Our proposed mechanism for implementing multiple FUSE threads is as follow:

1) Checks whether or not mount is finished. In case mount is not finished yes, it calls *fuse_get_mount()* function to execute FUSE mount system call.

2) Checks whether or not FUSE multi-threaded application is exited. While there are multiple threads will be used to get requests, there might be one thread gets request to exit (unmount). Thus, other FUSE thread should be

noticed that process is terminated.

3) Calls *readv()* system call on */dev/fuse* to receive request.

4) If request operation code is forget, it does not need to create new FUSE thread.

5) New FUSE thread is created only when there is no available thread is free to handle request and must be not greater than MAX number.

6) Then it executes corresponding *fuse_op* function pointer like fuse_init, fuse_getattr, fuse_write, fuse_read, etc depending on operation code.

Figure 4.3 shows the implementation of multiple FUSE threads. We build multiple FUSE threads instead of single FUSE thread in order to deal with the FUSE bottleneck problem on workload that user application makes a large number of small-file requests. FUSE threads should enable to perform as fast as user applications request and I/O worker threads perform. Generally, we could say that FUSE thread works as a bridge, which is built for making connection between user applications and I/O worker threads. When user applications make requests, they need FUSE thread to pass requests to I/O worker threads to handle requests. User application can make requests as many as they need, and I/O worker thread normally is the slowest worker in process of accessing data on file system. However, I/O worker thread is already implemented as multi-threaded application which can perform concurrent I/O operations as fast as they get requests. Therefore, the bottleneck is caused by a single FUSE thread since it works alone to receive requests from user applications, performs some FUSE operations, and forwards to I/O worker threads. So, why not building a bridge that can transfer many requests at the same time?

We implemented multiple FUSE threads to allow multiple requests made by user applications are enable to transfer to I/O worker threads. This can eliminate bottleneck caused by the original single FUSE thread and improve the overall performance. Our implementation ensures that the I/O performance is improved when we run GlusterFS on workload that user application makes a

lot of small-file requests or when many clients make request at the same to the same GlusteFS volume. However, depending on workload and hardware used, we do not recommend to use this FUSE multi-threaded application on large-file workload or small number of requests workload. On those workloads, FUSE multi-threaded application provides the same performance as single thread since there is no bottleneck at all, or performance can be slightly decreased since there is extra contention occurred between FUSE threads trying to access shared variables. Moreover, FUSE multi-threaded application should not be used when we are going to run system on hardware that consists few cores. Since GlusterFS is developed as multi-threaded program which already contains several threads plus when users use multi-threaded application to make requests, there must be not enough CPU to serve these working processes. The more multiple threads are used, then the more contention is increased.

In order to easily adapt the number of FUSE threads to the hardware environment and workload, we also added feature of FUSE thread to the configuration setting. Before running FUSE-based Gluster file system, user should be aware of hardware used and workload type. We provided this configuration setting on FUSE thread's feature to allow user to easily change the number of FUSE threads to be used. Similar to I/O worker threads, FUSE threads will be automatically created only when there are requesting still available waiting in queue and there is no available FUSE thread to receive request. However, when choosing to use multiple threads by setting number of FUSE thread greater than one, there must be at least two threads will be created even user application makes a single request. Due to the implementation design coding in figure 5 above, when the first thread received request then it will check whether there is other thread available or not. In this case, no FUSE thread is available since it has only one thread, then it will call *fuse_start_thread* function to create another new FUSE thread.

Chapter 5

Evaluation

Our goal of this evaluation is to see the performance and CPU utilization caused by FUSE-based GlusterFS which FUSE thread was implemented as multiple threads versus the original single FUSE thread.

5.1 Experimental Setup

	Specification
CPU	two Intel(R) Xeon E5-2650 v4 @ 2.20GHz
Core	24 physical cores (48-HT)
RAM	32GB
Storage	256GB SAMSUNG 850 PRO SSD
Network	40G/s Ethernet

Table 5.1 Hardware Specification

Our tests were performed on three machines, which one is served as client node, and the other two are served as server nodes. Those machines contain two Intel Xeon E5-2650 v4 with 12 cores for each and running at 2.20 GHz.

Due to Hyperthreading feature is enable, each node consists 48 cores. 32GB of main memory is resided in each node, but our experiments are not going to test with caching performance because we just want to see the real I/O performance on disk storage. The storage device we used for these experiments is 256GB of Samsung SSD 850 Pro. The operating system was Linux Centos 7.3.1611, with kernel version 4.9.16. Client and server nodes are connected through TCP/IP with the maximum network speed is 40Gb/s, clustered on Gluster file system version glusterfs-3-9-0, while the version of user-space FUSE library was 3.0.1. The trusted storage pool on GlusterFS server nodes is mounted on XFS file system. Two GlusterFS volumes, distributed volume (number of bricks: 2) and replicated volume (number of bricks: 2 x 1), are used for these experiments.

5.2 Tested Configurations

Configurations on each GlusterFS volume are set as follow:

Option	Value
server.event-threads	2
client.event-threads	2
performance.io-thread-count	6
performance.client-io-threads	on
performance.readdir-ahead	on
performance.read-ahead	off
performance.write-behind	off
performance.io-cache	off

Table 5.2 Configuration setting on GluserFS's Distributed volumen and Replicated volume.

5.3 Benchmark Methodology

The benchmark we used to measure the performance of these experiments was fio benchmark [24]. Fio is stand for Flexible I/O, written by Jens Axboe to enable flexible testing of file operation I/O performance on various file systems and schedulers. By using Fio benchmark, we tested GlusterFS performance by setting various options. We tested on both small-file 4K, and large-file 1M workload with direct I/O, io-engine is sync / libaio with various iodepth from 1 to 32, and numjobs (user requests) from 1 to 32.

5.4 Benchmark Results

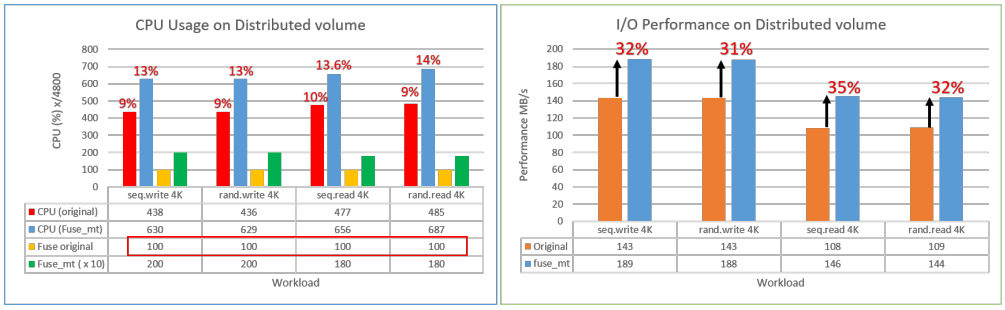

Figure 5.1 CPU usage and I/O performance of 4KB chunk size workload on Distributed volume.

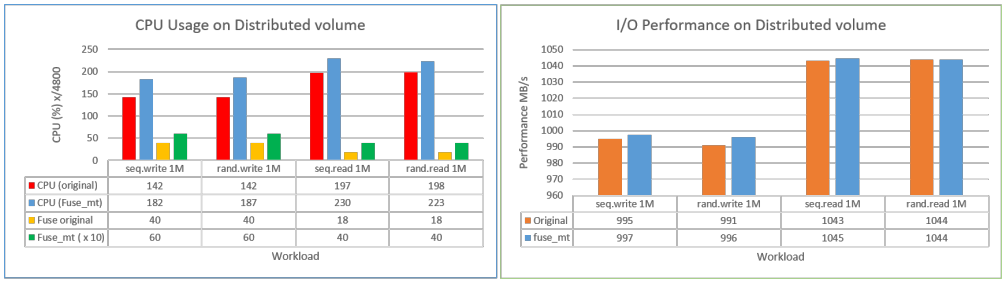

Figure 5.2 CPU usage and I/O performance of 1MB chunk size workload on Distributed volume.

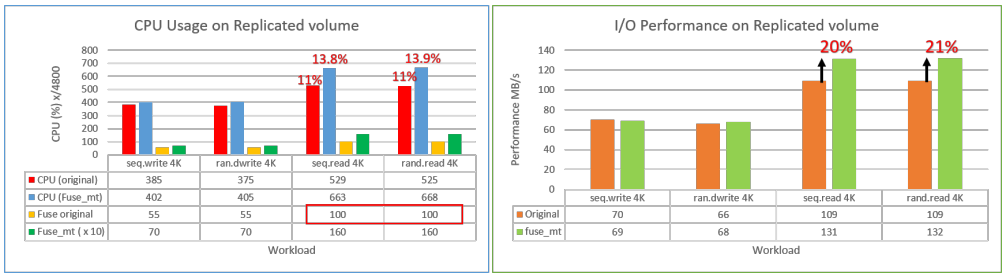

Figure 5.3 CPU usage and I/O performance of 4KB chunk size workload on Replicated volume.

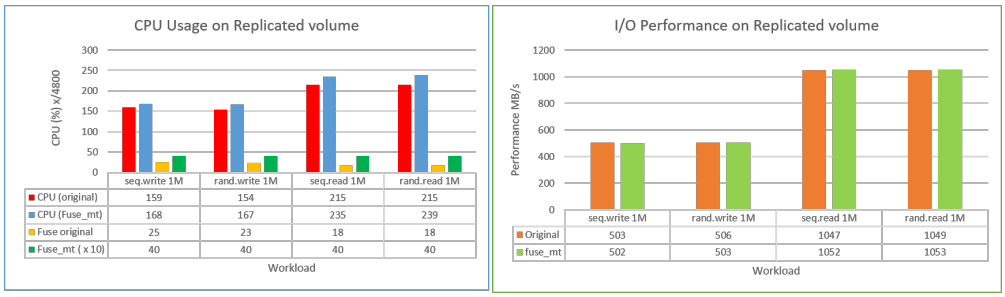

Figure 5.4 CPU usage and I/O performance of 1MB chunk size workload on Replicated volume.

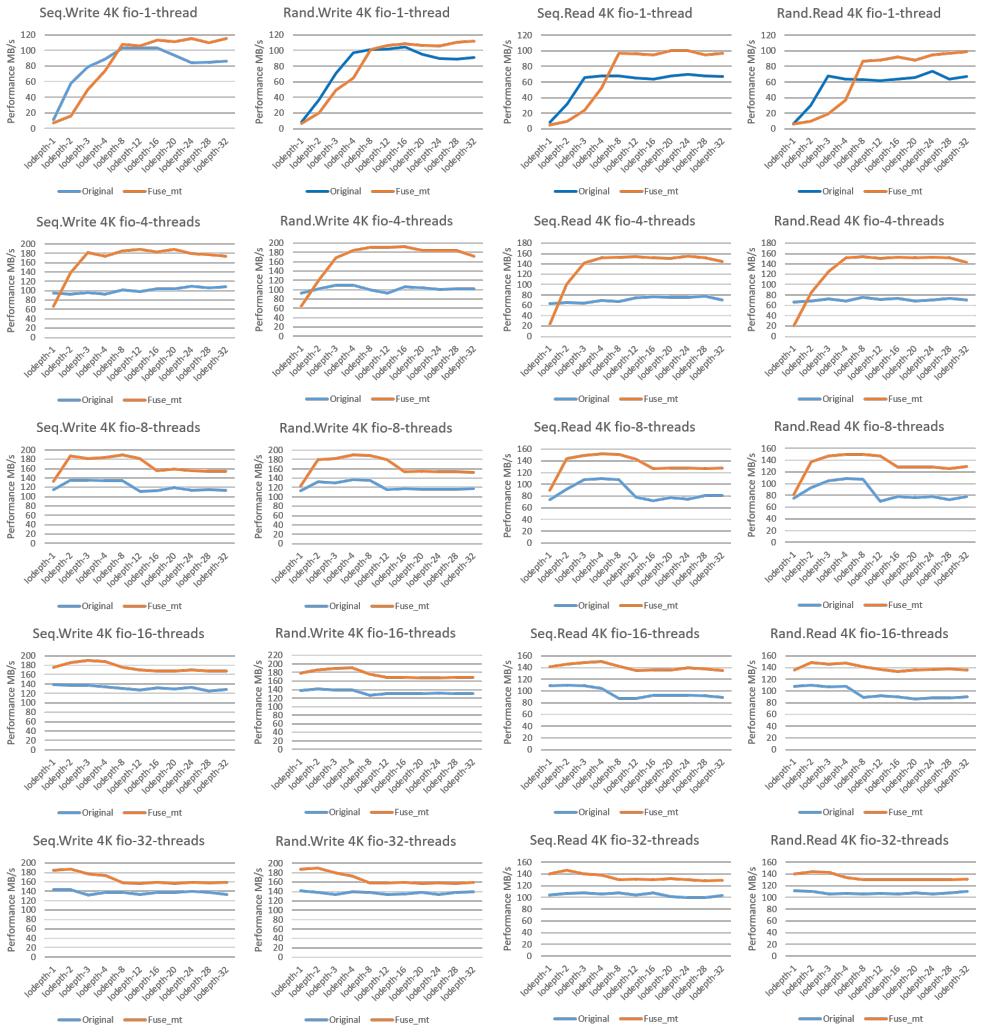

Figure 5.5 Benchmark results of asynchronous I/O performance of 4KB chunk size workload on Distributed volume with various number of numjob and iodepth.

Chapter 6

Discussion

In this section, we are going to discuss about the performance result of our experiments on GlusterFS with multiple FUSE threads and the original single FUSE thread, which tested with various workloads. We performed all benchmarks for three times and then calculated the average value for being used in these experiments. For all of our experiments we did not include caching performance, so we performed all benchmarks by using direct I/O feature.

Figure 5.1 to figure 5.4 shows the performance results of synchronous I/O with 32-thread numjobs by running Fio benchmark as user application making requests to GlusterFS's volumes. On Figure 5.1 indicates the CPU usage and I/O performance result tested on small-file 4KB workload performed by Fio benchmark on GlusterFS's Distributed volume. We can see that when running on GlusterFS with a single FUSE thread, FUSE thread used CPU up to 100%. This 100% CPU usage can result in bottleneck caused by FUSE thread, which is absolutely busy to handle requests. This means that there are lots of requests made by user applications and requests are waiting in FUSE queue, which cannot be picked to up handle as fast as they arrived. No matter how fast the I/O worker threads perform I/O operations, requests are not quickly sent by FUSE

thread to I/O worker threads. Therefore, the performance is unable to reach the maximum value. As what we expected, when we used multiple thread instead the original single thread, we can see that the CPU usage of FUSE threads is increased. As the result, the performance of both sequential and random write workloads is improved up to 32%, while the performance of sequential and random read workload is improved up to 35% and 32% respectively.

Similarly, when user application performed 4KB chunk size requests on GlusterFS's Replicated volume, we can see from the figure 5.3 that performance of reading 4KB workload is improved by 21% compare to the original single FUSE thread. We can see from the graph of CPU usage when a large number of 4KB chunk size read workload requests is performed by Fio benchmark user application, the original single FUSE thread used CPU up to 100%. When this bottleneck is eliminated by using multiple FUSE threads instead, we can see that CPU usage is also increased, and performed is improved as expected. Yet not every workload is increased if there is no bottleneck at all. Like what we can see from the performance result of writing workload, even though 4KB chunk size requests are performed, I/O performance is not improved. This is not because of FUSE thread, but it because I/O worker threads need to write data twice per request. Performance of writing is also twice less than performance of writing workload on Distributed volume. This is the disadvantage of Replicated volume, which needs to save duplicated data to ensure the reliability. However, when reading data from Replicated volume, no need to read twice like writing, it will perform by reading from only one sub volume.

However, we also expected that performance is not going to be increased when testing on large-file workload. As what we can see in figure 5.2 and figure 5.4, when user application's requests are large-file 1MB chunk size workload, the single FUSE thread performed well. Unlike 4KB chunk size workload, which used CPU up to 100%, so the I/O performance are not somehow improved. Both multiple FUSE thread and single FUSE thread, they produce the similar

performance result, while the CPU usage of multiple FUSE threads is a little bit increased since there are lock contention occurred between those FUSE threads. Therefore, it is not better to use FUSE multi-threaded application when running on large-file workload or a small number of requests.

Figure 5.5 shows the benchmark result of asynchronous I/O performance of small-file 4KB chunk size workload on Distributed volume tested with different various number of numjobs and iodepth. Like what we have mentioned above, when user application just performed only few requests, the original single FUSE thread can perform better than multiple FUSE threads. However, once when we increased the number of iodepth to enable user application makes request without waiting result back, then the number of requests is increased. The more number of iodepth is increased, the more number of requests is also increased. Thus, we can see that the performance of multiple FUSE threads keeps improving and produces the results better than the original single FUSE thread.

However, even though we kept increasing the number of iodepth together with number of numjobs, performance is limited, which can be improved until the maximum value only. In this paper, we focused only on the bottleneck caused by single FUSE thread. We have implemented this FUSE multi-threaded application inside GlusterFS in order to allow FUSE thread to perform as quickly as the I/O worker threads can do.

Chapter 7

Related Work

Since FUSE is a popular framework for implementing user-space file system, many distributed file systems used FUSE as a data accessing protocol. However, we need to pay a little attention to understand how FUSE is design and its performance. Especially, we have to be aware of which workload and hardware used when we want to build multi-threaded FUSE application instead of single thread. Many researchers have been working on file system and FUSE to improve more several benefits. In case of our knowledge, we have researched on several papers to gain more ideas to do our work and found that two papers are really good for our study.

The first paper we have studied was about *"Performance and Extension of User Space File System"* [22], which written by *Aditya Rajgarhia and Ashish Gehani*, published in 2010. Their work was mostly focused on the evaluation of FUSE performance by using Java binding library wrappers. They performed their experiments on FUSE version 2.8.0-pre1, which is released in 2008. Their paper conclusion was mentioned that it is a good idea to use FUSE on the system that run as Web servers which contains a lot of small-file metadata operations, and other systems that typically deal with small files and several

clients accessing data on the same servers. They also mentioned that it is better to use FUSE for personal computers and small-scale servers. Grid applications and multimedia servers are also encouraged to use FUSE since they perform large-file I/O transfers.

The second paper was about *"To FUSE or NOT to FUSE: Performance of User-space File Systems"* [23], which was presented at USENIX 15th conference on File and Storage Technologies on February 2017 by *Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok*. The main purpose of this paper was to show that in which situation that FUSE should be and should no be used. They have explained the detailed design of FUSE and performed many experiments to conduct a broad performance characterization of FUSE. They also mentioned the comparison between FUSE and native ext4 file system and make some conclusions that FUSE can perform within 5% of native Ext4, but some unfriendly workloads FUSE degrades their performance by up to 83%.

Chapter 8

Conclusion

Even though FUSE-based GlusterFS is the highly scalable file system which can provide the reliability and higher performance, but it cannot be suitable for all workloads and environment hardware used. We need to make some changes and adjust configuration setting on our corresponding workloads and hardware used. Like in our paper, we have presented the general knowledge of distributed system, Gluster file system, and the detailed design of FUSE. We have implemented multi-threaded FUSE application on GlusterFS to ensure that there is no more bottleneck caused by FUSE thread. This bottleneck elimination resulted in performance upgradation for the whole system. As what we have mentioned above, at least 35% is improved on small-file reading workload, and 32% is improved on small-file writing workload when compared to the original single FUSE thread.

Chapter 9

Future work

The bottleneck caused by FUSE thread is eliminated by implementing multi-threaded FUSE application is used instead of single thread on GlusterFS. However, even though the performance is improved, but the it is not as high as the performance tested on local SSD storage device. There should be the performance overhead of FUSE as what we have mentioned in above section. Before using FUSE, we need to take care of two trade-off factors (1) how large is the performance overhead cause by user-space implementation and (2) how much easier is it to develop user-space file system. We plan to find out those possible problems and investigate the performance on GlusterFS's server side as well. We are going to reduce unnecessary works as much as possible like merging FUSE threads and I/O worker threads together. A merged FUSE thread and I/O worker thread will receive request from FUSE queue and perform I/O operation directly without accessing put/get to/from I/O operation queue. This means we are going to remove I/O operation queue used by FUSE threads and I/O threads. We do not expect too much performance improvement since context switch overhead of FUSE is too high. However, this could reduce CPU usage of the whole system.

Bibliography

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 20–43.

[2] P. Schwan, "Lustre: Building a File System for 1,000-node Clusters," in *Proceedings of the 2003 Linux Symposium*, 2003.

[3] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 307–320.

[4] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008, pp. 17–33.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System, "The Google File System," in *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies*, 2010.

[6] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm Grid File System," in *New Generation Computing*, vol. 28, no. 3, pp. 257–275, 2010.

[7] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, M. Seaman, and D. Subhraveti, "GPFS-SNC: An enterprise storage framework for virtual-machine clouds," in *IBM Journal of Research and Development*, vol. 55, no. 6, 2011.

[8] GlusterFS, http://www.gluster.org

[9] Parallel Virtual File System (PVFS), http://www.pvfs.org

[10] OrangeFS, http://www.orangefs.org

[11] BeeGFS, https://www.beegfs.io

[12] Infinit, https://www.infinit.sh

[13] ObjectiveFS, https://www.objectivefs.com

[14] MooseFS, https://www.moosefs.com

[15] Quantcast File System, https://github.com/quantcast/qfs

[16] OpenAFS, http://www.openafs.org

[17] Tahoe-LAFS, https://tahoe-lafs.org/trac/tahoe-lafs

[18] XtreemFS, http://www.xtreemfs.org

[19] Maarten van Steen, Andrew S. Tanenbaumi, "A brief introduction to distributed systems," in *Computing*, vol. 98, Issue 10, October 2016, pp. 967–1009.

[20] M. Szeredi, "Filesystem in Userspace," http://fuse.sourceforge.net, February 2005.

[21] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, Osamu Tatebe, "Optimizing Local File Accesses for FUSE-based Distributed Storage," in *SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012.104

[22] A. Rajgarhia and A. Gehani, "Performance and Extension of User Space File Systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 206-213

[23] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok, "To FUSE or NOT to FUSE: Performance of User-Space File Systems," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, 2017

[24] Jens Axboe, "TFio benchmark," https://github.com/axboe/fio, 2006

요약

유저 스페이스 파일 시스템(FUSE)은 많은 분산 파일 시스템에서 데이터 접근 프로토콜로 사용되고 있다. 퓨즈는 유저 애플리케이션으로 부터 데이터를 전달받아 I/O를 위한 애플리케이션으로 데이터를 전달하는 역할을 한다. 하지만 분산 파일 시스템에서 퓨즈를 사용하는 경우 다수의 작은 파일 혹은 많은 클라이언트를 사용하는 상황에서 매우 큰 오버헤드를 가진다. 이러한 오버헤드는 시스템 전체에서 bottleneck이 되어 성능을 크게 감소시킨다. 본 논문에서는 퓨즈가 싱글 쓰레드로 동작하기 때문에 발생하는 병목 현상을 없애기 위해 멀티 쓰레드 퓨즈를 구현한다. 이 방법을 통해 퓨즈 큐에 데이터가 존재하면 새로운 퓨즈 쓰레드가 생성되어 해당 데이터를 병렬적으로 처리하게 된다. 이러한 방법을 대표적인 분산 파일 시스템인 GlusterFS에 구현한 결과, small file write에서 32%의 성능 향상을 보였고 small file read에서 35%의 성능 향상을 보여 주었다.

주요어: 퓨즈 병목, 멀티쓰레드 퓨즈, GlusterFS

학번: 2015-23300