M.S. THESIS

# Per-core file allocation method for eliminating the overhead of maintaining consistency between nodes in a distributed file system

분산 파일 시스템에서 노드 간 컨시스턴시 유지 오버헤드의 제거를 위한 per-core file allocation 방법

AUGUST 2017

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Woohyung Han

M.S. THESIS

# Per-core file allocation method for eliminating the overhead of maintaining consistency between nodes in a distributed file system

분산 파일 시스템에서 노드 간 컨시스턴시 유지 오버헤드의 제거를 위한 per-core file allocation 방법

AUGUST 2017

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Woohyung Han

# Per-core file allocation method for eliminating the overhead of maintaining consistency between nodes in a distributed file system

# 분산 파일 시스템에서 노드 간 컨시스턴시 유지 오버헤드의 제거를 위한 per-core file allocation 방법

지도교수 염헌영

이 논문을 공학석사학위논문으로 제출함

2017 년 06 월


서울대학교

컴퓨터 공학부

한우형


한우형의 석사학위논문을 인준함

2017 년 07 월

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

# Abstract

## Per-core file allocation method for eliminating the overhead of maintaining consistency between nodes in a distributed file system

Woohyung Han
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

Distributed File System (DFS) is a file system that allows access to multiple storage servers through a computer network. The modern DFS offers a variety of functions such as load balancing, location transparency, high availability, and fault tolerance. Among them, the fault tolerance is one of the most important functions required to protect data from server and disk failures. GlusterFS of a typical DFS supports replication, which replicates data for fault tolerance and stores it on a separate server, and Erasure code that stores the parity on another sever after encoding the data. When it comes to these two ways, it is essential to maintain the data consistency between the server nodes in order to maintain the consistency of the data because the information about one data is distributed and stored across multiple server nodes. If the data consistency is not maintained, each server node stores data with different contents, which leads to the destruction of fault tolerance. Therefore, the GlusterFS uses a method to ac-

quire a lock in all servers when performing each operation to solve the problem. The reason for using this method is because file operations can be delivered as intermixed between sever nodes. All file operations must be atomically applied to the entire sever node. However, in a current implementation of the GlusterFS, it can be operated in parallel in multiple io-thread and event-thread even in operations on the same file, so that it requires a concurrency control. This can cause up to two additional round trips as well as overheads such as managing locks. Therefore, we propose a method to maintain data consistency between server nodes without an additional concurrency control by keeping the order of operations on the same file in the whole system by making the operations of the same file performed on the same core all the time. In this way, we could achieve mean 63% and up to 83% performance improvements in randread, and mean 60% and up to 69% performance improvements in randwrite.

**Keywords**: GlusterFS, Erasure Code, Cluster Lock

**Student Number**: 2015-21276

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As the era of cloud and big data arrives, data storage is increasing dramatically. Data up to 4.4ZB in 2013 is expected to be dramatically increased to up to 44ZB in 2020[6]. Such data must be stored somewhere, leading to the development and advancement of the Distributed File System. The Distributed File System (DFS) consists of client and server and provides users with various levels of transparency to help them access to distributed storage and shared data. The DFS has been steadily developed since the introduction of the Network File System (NFS) in the 1970s, and today, various DFSs such as Ceph[2], GlusterFS[1], HDFS[9], and Lustreref10 have been developed. These DFSs support methods such as replicating files to multiple sever nodes or storing an additional parity encoded to protect data from disk and sever failures. However, in this situation, it is essential to maintain data consistency between servers due to the fact that multiple servers must store data related to each other. If there is no concurrency control method for this, multiple servers will store different data, which can make it impossible to recover the data. Therefore, GlusterFS, one of the representative distributed file systems, uses a method to acquire a lock in all servers when performing each operation to solve the problem. However,

this method is very inefficient because the atomicity of the file operation can be guaranteed, but an additional round trip is occurred to acquire a lock, and the overhead of managing the lock is added at the same time. In here, we will talk about a method to maintain data consistency while eliminating the locks that occur on every file operations through the structure that guarantees the order of operations in the entire system by always allocating operations on the same file to the same core to solve the problem. In this way, we could achieve mean 63% and up to 83% performance improvements in randread, and mean 60% and up to 69% performance improvements in randwrite.

# Chapter 2

# Background & Motivation

GlusterFS[1] is a scale-out network-attached distributed file system provided by Redhat as open source. The GlusterFS consist of a stackable and modular layer called a translator, and each translator provides independent functions such as mirroring, replication, erasure coding, io caching, read-ahead, and write-behind. Among them, the erasure coding translator, provided by the client, provides functions of data failure recovery and high availability by storing additional data redundancy in a separate server. To ensure that the stored data is correct, the data consistency stored on each server node must be guaranteed. However, in the GlusterFS, it can be performed simultaneously on multiple cores even if there is access to the same file, so that an additional concurrency control is required to ensure the data consistency. The Ceph[2], type of a distributed file system, provides as open source, uses a method to lock an object called pg on the primary node, but the method can't be used in the GlusterFS that there is no primary node. Therefore, the GlusterFS now uses a method to directly lock files on all server nodes. This method has the problem that the latency and the CPU utilization of the server increase because it accesses to all server nodes every operation although the data consistency can be maintained. The optimization

such as eager-locking has been applied to solve the problem, but there is still overhead such as lock list management. Therefore, this paper proposes a method to remove the overhead while maintaining the data consistency by guaranteeing the order of file operations on client and server and locking it only when opening the file instead of locking it every operation.

## 2.1 Dispersed Volume

Dispersed Volume is data storage methods that provide space-efficiency as well as fault tolerance for storage or sever failure based on Erasure Code (EC). The data is divided into the n parts of a chunk which has 512Byte, and these data are encoded to produce k additional parity. Now, n+k data are stored in each server, and when actual data is needed, original data can be recovered if only n chunks are read and decoded. In this case, there is no problem even if a maximum of k data is lost. The GlusterFS uses a Reed-Solomon algorithm in the encoding and decoding process, which is a Non-Systematic algorithm in which data and parity are mixed and stored instead of a Systematic method in which data and parity are stored separately. However, in this paper, figure was expressed in a Systematic form for the convenience of explanation.

Figure 2.1 shows a writing process in 2+1 Dispersed Volume. The encoding and decoding process of the EC is performed on the client. The data on the client node is divided into two chunks (1024 Bytes) because of the 2+1 configuration, and these two chunks are encoded, resulting in three chunks by adding parity. When performing the encoding process, padding is added if the size of the data is smaller than 1024 bytes. The three encoded chunks are transmitted to each server node through the network.

Figure 2.2 shows a reading process in 2+1 Dispersed Volume. In the 2+1 configuration, original data can be recovered if only two chunks are read. Therefore, the client node selects two server nodes for reading the data. The choice is followed by the read policy, which is provided as an option in the GlusterFS.

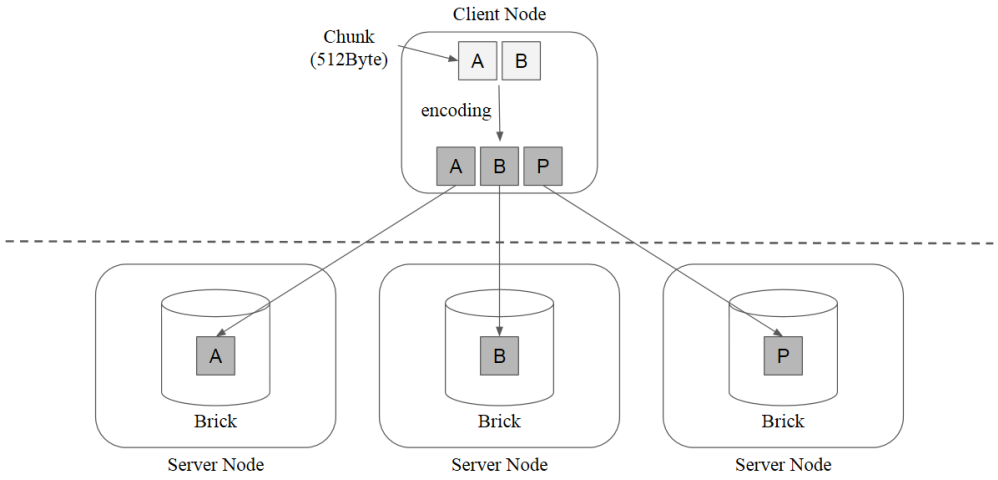


Figure 2.1 Writing process in 2+1 Dispersed Volume

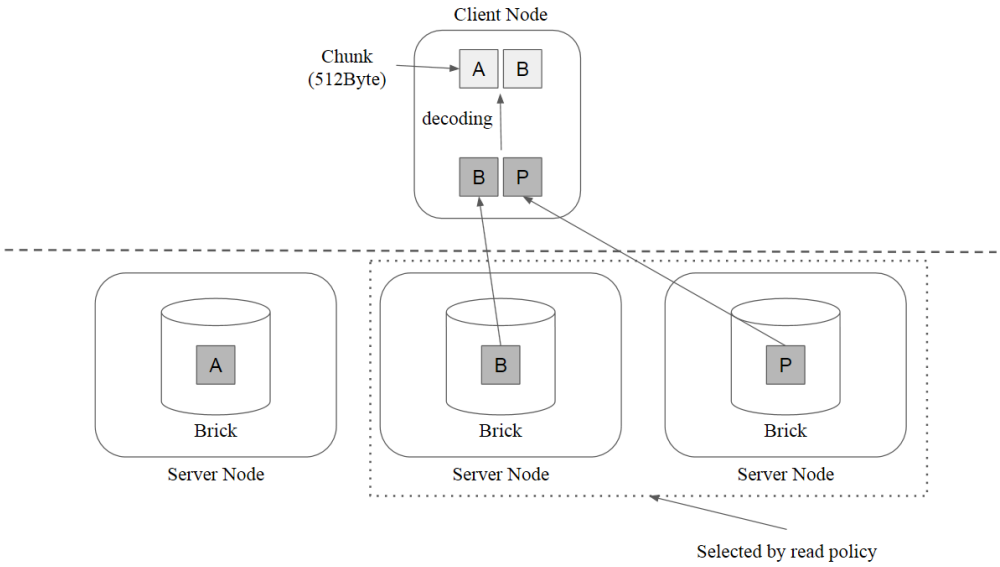


Figure 2.2 Reading process in 2+1 Dispersed Volume

Currently, the GlusterFS supports a round robin method that selects all the server nodes alternately, and a hashing method that always selects the node by hashing the information of the file. After selecting two nodes from the client node, the selected server node transmits the stored chunk to the client over the network. The client goes through the decoding process of the received two chunks and returns the result.

## 2.2 The maintenance of Data Consistency in Dispersed Volume

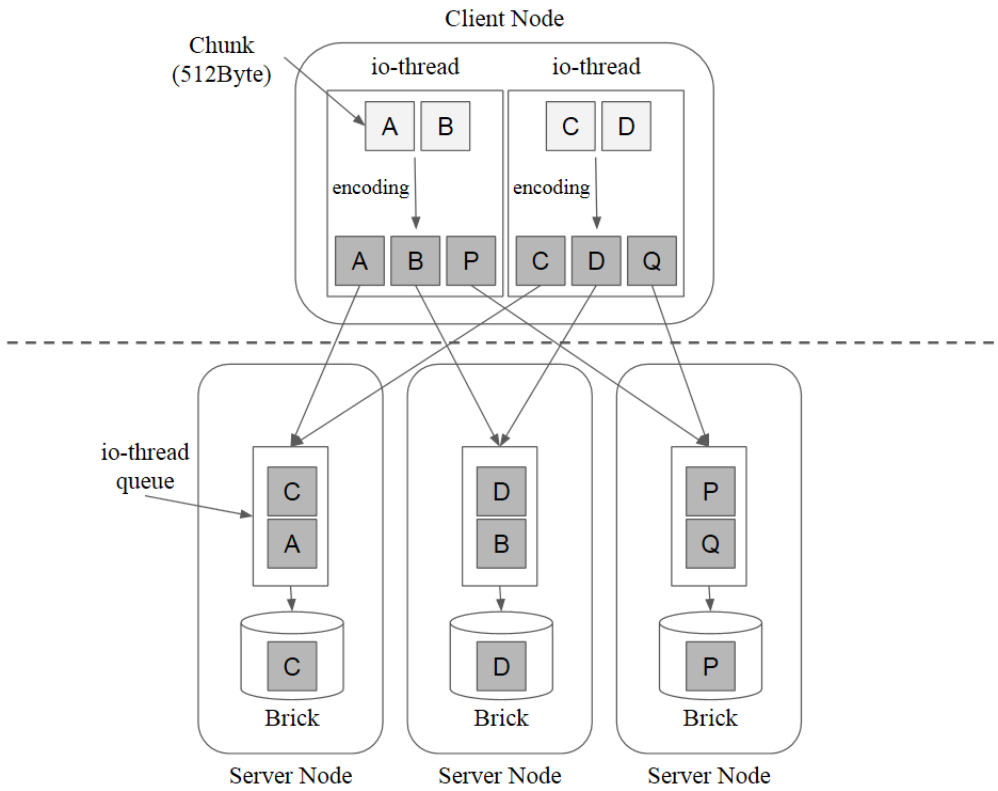


Figure 2.3 Example of Data Consistency broken in 2+1 Dispersed Volume

Data is segmented and then stored in each server node in Dispersed Volume, and in this case, it is essential to maintain the data consistency. If the

client performs multiple write processes to the same file without a particular concurrency control, the data consistency may be broken. In the case of figure 2.3 means consecutive writes, it shows that the data is stored incorrectly as the data consistency is broken. First, if two writes are performed on the client, these operations are performed concurrently by the io-thread. Each fragment is encoded concurrently and then transmitted over the network. The transmitted chunks are enqueued into the io-thread queue by the event-thread, and then io-threads of sever nodes are dequeued from the queue, performing a task to store them in the actual storage. However, there are some possibilities that chunks stored in each server can be mixed and stored in this process. First, as the io-thread is concurrently operated on the client, it is not true that A, B and P are transmitted, and then C, D and Q are transmitted. However, it is possible that A and B are transmitted, then C, D, and Q are transmitted, and then P is transmitted. In this situation, the server can't distinguish whether the parity for C and D is P or Q. Next, the same problem can occur because the event-thread receiving from the network on the server and the io-thread performing I/O at the actual storage are concurrently operated. The above figure 2.3 shows an example that A, B and P, or C, D and Q should be stored in each server node, but C, D and P are actually stored. In this case, the wrong data can be retrieved when reading them. This problem does not only occur between writing and writing, but also between reading and writing. Accordingly, both reading and writing require a concurrent control.

## 2.3 Dispersed Volume Cluster Lock

The GlusterFS uses a method to acquire a lock in all server nodes in order to maintain data consistency. Figure 2.4 shows the sequence of acquiring a lock in all server nodes. The lock process consists of two sub-phases. First, the client requests broadcasting non-blocking lock to all servers. If all servers acquire a lock successfully, the lock process is terminated. However, if even one server

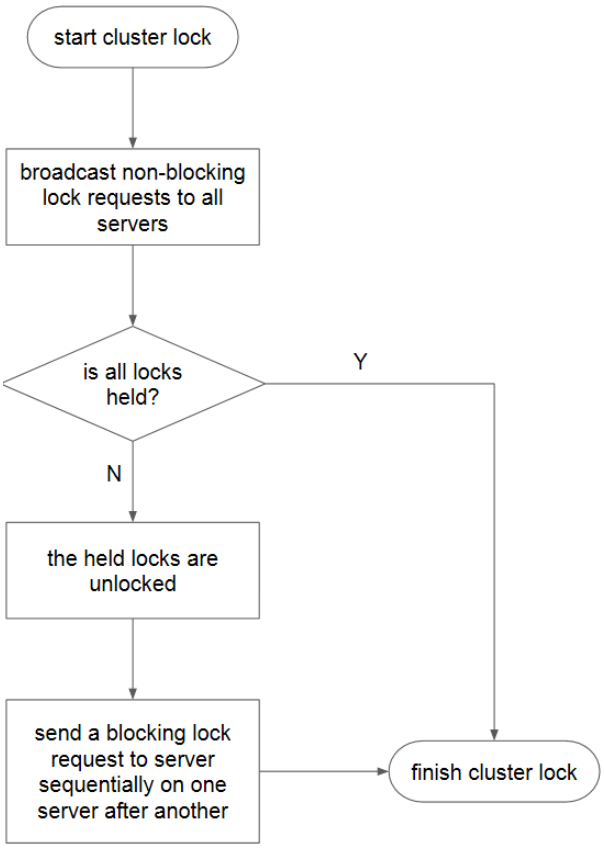


Figure 2.4 Dispersed Volume Lock Flowchart

can't acquire a lock, first unlock all the locks that have been acquired so far, and carry out the second phase. In the second phase, it acquires a blocking lock of server one by one in order. This process is free of deadlock because it always proceeds in the same order. When acquiring a lock of all servers, the lock process is terminated. This is done every time when a file operation is performed. Therefore, there is a problem that a maximum of two round trips may occur per a file operation. The GlusterFS may eliminate the round trips with optimizations such as eager-locking to solve the problem, but still has a large overhead in managing the lock list, and in the case of sporadic workloads, performance is rather reduced.

## 2.4 Structure of current GlusterFS

The reason for using this inefficient method in the GlusterFS is because file operations can be delivered as intermixed between sever nodes. The io-thread and event-thread running on the client and server nodes can't guarantee the atomicity of file operation because they are concurrently executed on multiple cores in parallel. Figure 2.5 shows the structure of this current GlusterFS. First, applications on the client deliver the file operation to the iot-queue. Contention by lock occurs because one io-thread queue exists globally for each node. Queued operations are dequeued by the io-thread running in parallel. Therefore, there is a possibility that several operations are mixed and transmitted between nodes in this process. Next, the io-thread performs an encoding for erasure code and a translator for other GlusterFS and transfers it to the network through the socket. The transferred operation is handled by the event-thread on the server. In the event-thread too, operations can be mixed depending on the scheduling method because multiple threads operate in parallel on multiple cores. The event-thread enqueues the operation to the global iot-queue of the server, and the io-thread of the server dequeues the operation, leading to actually perform I/O on the storage. This structure is a way to maximize parallelism in a stripe

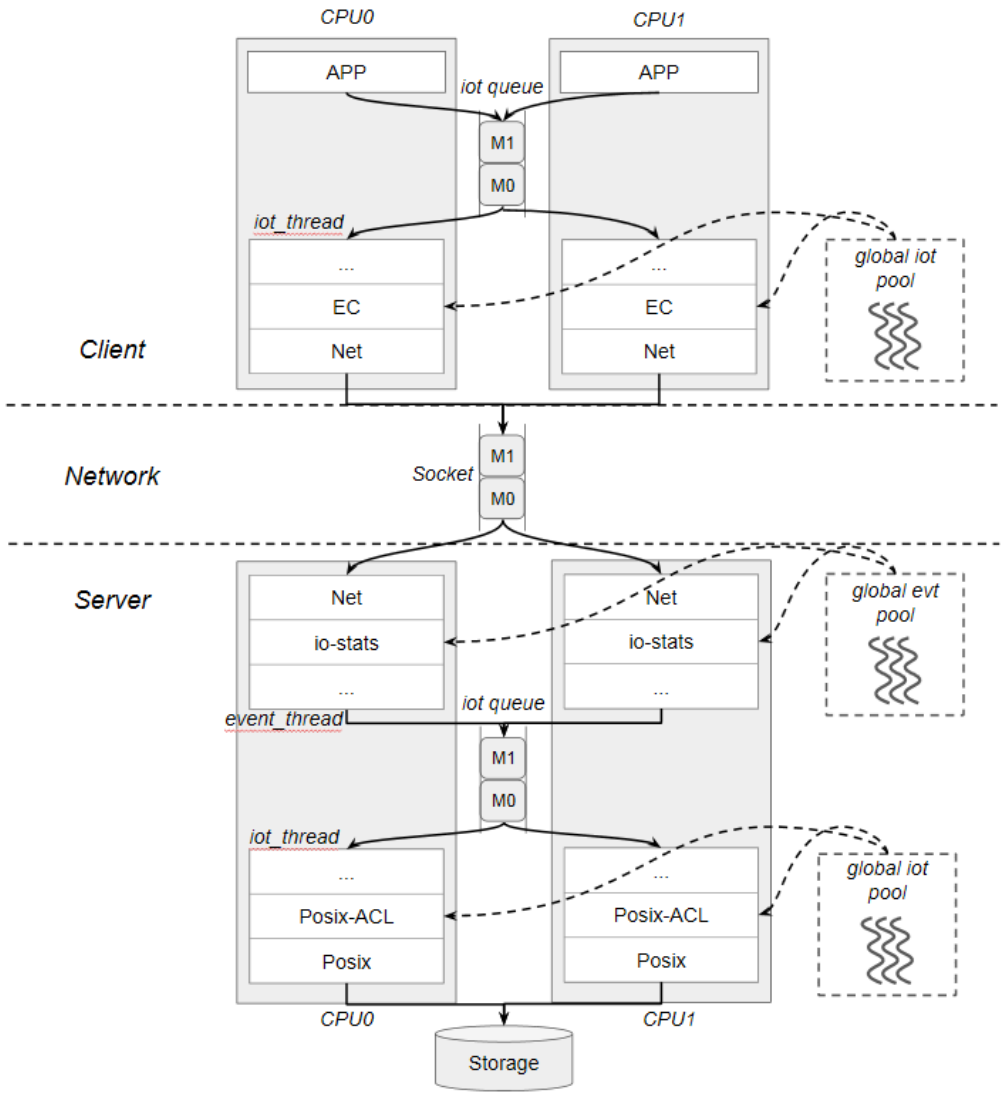


Figure 2.5 current GlusterFS's client and server structure

that does not require the maintenance of data consistency, but in the case of replication or erasure code, even if many operations are sent to several cores, serialization must be performed in order to maintain the data consistency, so there are problems in which it is not only inefficient, but also requires additional overheads for serialization. Therefore, this paper proposes a method to maximize parallelism by automatically serializing data without additional overheads by restricting the whole operations on the same file to only one core.

# Chapter 3

# Design & Implementation

It needs to serialize operations on the same file in order to remove a cluster lock. This can be done by always performing the operations on the same file in the same core. In the existing GlusterFS, even operations on the same file can be scheduled on each core, but now the operations on the same file are always schedules in the same core, so that it stops the operations on one file from being performed on multiple cores. Figure 3.1 shows such a system.

Per-core file allocation: In a conventional GlusterFS structure, it could be operated on multiple cores if there is no lock, even in operations on a single file. This made serialization by locks essential. However, by scheduling operations on the same file to run on the same core, it makes it possible to serialize file operations without any locks.

Gfid-queue table: Gfid is a value to uniquely characterize a file in the GlusterFS. I t can be used to distinguish which core a particular file will be mapped to. Once operations on a new file start, first search gfid-queue table. If the table does not yet have a corresponding gifd value, assign a new core to the round robin policy. In the future, all operations corresponding to the gifd will be performed in the corresponding core.

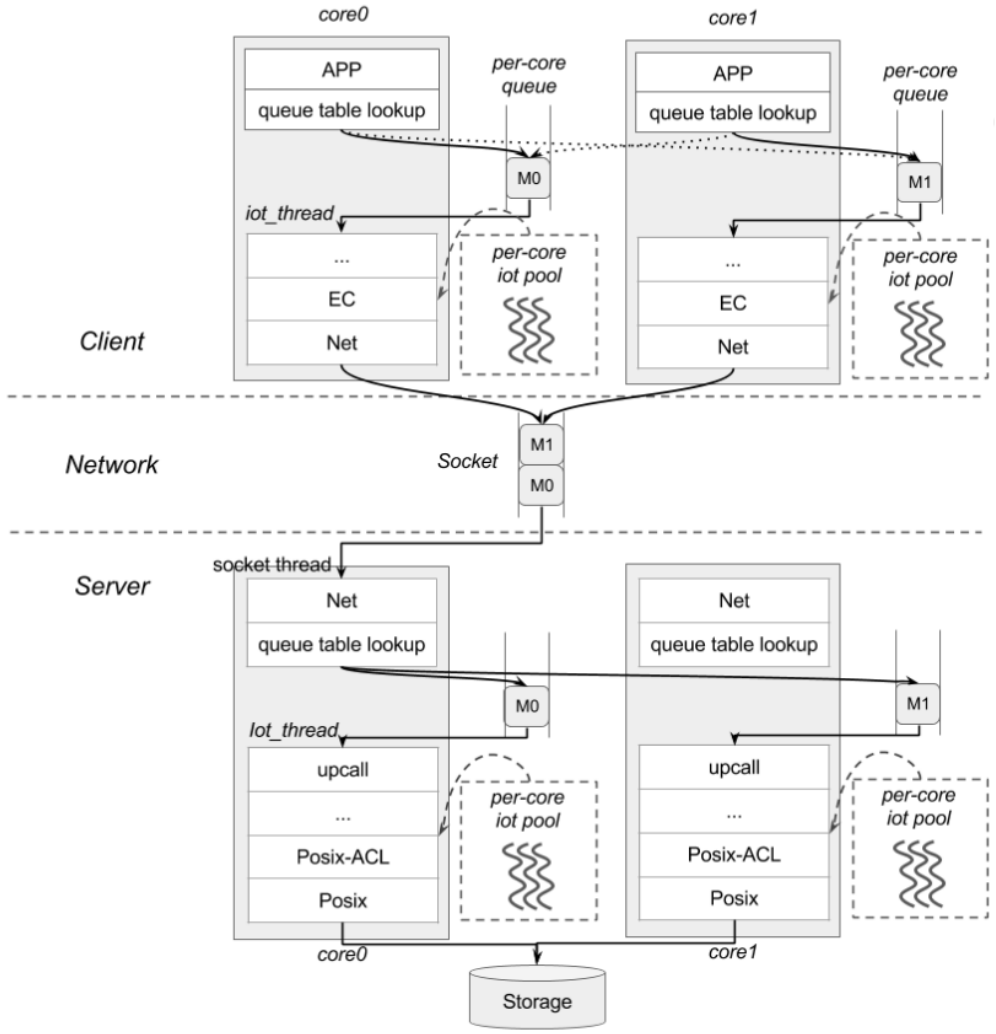


Figure 3.1 Suggested GlusterFS structure

Per-core lock-free queue: There is no longer a global queue, and a thread is pinned to each core, so no more locks on the queue are needed. This prevents performance deterioration due to a lock contention.

Per-client socket thread: In the past, many event-threads exist, and the event-thread that is scheduled regardless of the client performed the operation. However, the event-thread for one client must be limited to one in order to maintain the order of file operations. This problem can cause scalability to be decreased as clients increase. Therefore, the problem can be solved by creating some socket threads and taking full charge of specific clients.

# Chapter 4

# Evaluation

| | Specification |
|---|---|
| CPU | Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz |
| RAM | 32GB |
| Storage | SAMSUNG 850 PRO SSD |
| Network | 10G Ethernet |

Table 4.1 Hardware Specification

We compared the newly implemented version by removing the lock through the proposed structure with the GlusterFS using the existing cluster lock. The specifications of server and client are shown in Table 4.1. For the performance evaluation, we used FIO[4] 2.17 version and measured the bandwidth change with Direct I/O and block size 4K 256K. The GlusterFS uses 3.9.0 version and is configured with 2+1 EC (1 redundancy) and enables the client-io thread option.

Figure 4.1 and 4.2 show the difference in randread and randwrite performances of original and no-lock according to block size. There were mean 63%

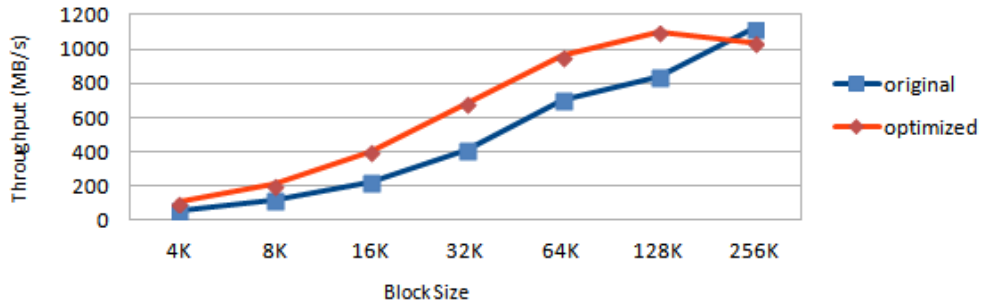


Figure 4.1 Comparison of randread performance according to block size

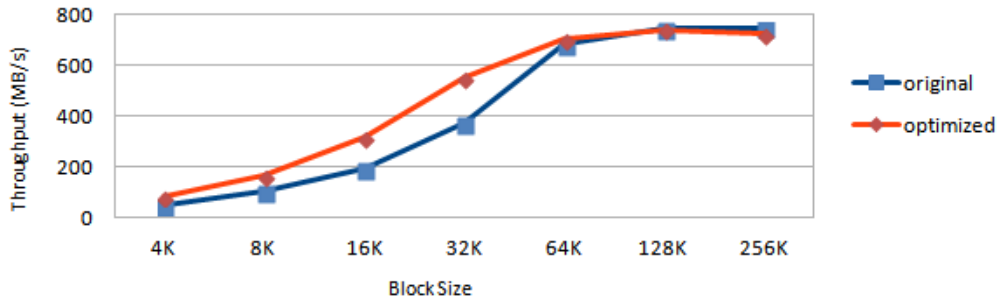


Figure 4.2 Comparison of randwrite performance according to block size

and up to 83% performance improvements in the randread, and mean 60% and up to 69% performance improvements in the randwrite. In the randread, the performance of no-lock is reduced at 256K, which seems to be because of coming to the maximum performance of 10G Ethernet. In fact, performance has decreased after 256K, which reaches the maximum performance of 10G Ethernet, in the original. On the other hand, in the randwrite, the performance did not increase at 740MB/s, which also seems to be because of coming to the maximum performance of 10G Ethernet. When trying to use chunks in erasure code, three chunks including parity should be used in practice, so it can only have the performance of 740MB/s which is 2/3 of a maximum bandwidth.

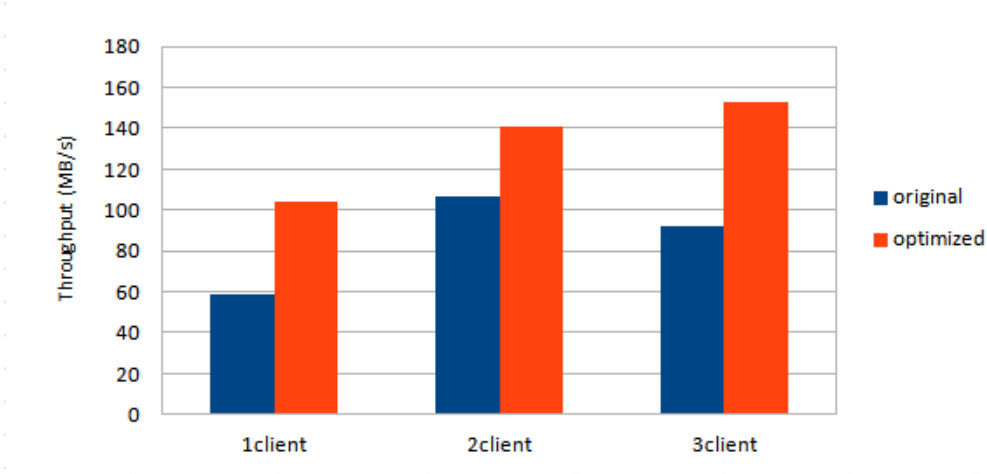


Figure 4.3 Performance comparison based on the number of clients

Figure 4.3 shows the total performance according to increasing the number of clients in the same server environment, as well as the CPU usage of socket threads on each server. And, it was found that the scalability is improved by 153MB/s from three clients in no-lock unlike the original that reaches about 107MB/s from two clients.

# Chapter 5

# Conclusion

We discussed how to allocate to a core on a file-by-file basis in order to eliminate the overhead of acquiring a lock on every file operations in the Dispersed Volume of the GlusterFS. The cluster lock was necessary because jobs could be assigned to different core, even in operations on the same file, in the existing GlusterFS. However, in the proposed structure, operations on the same file are always assigned to the same core, so no additional serialization is required. Through this implementation, we could achieve the performance improvements of mean 63% and up to 83% in the randread and of mean 60% and up to 69% performance improvements in the randwrite. And, experimental results have shown us that the scalability increases as the number of clients increases.

# Bibliography

[1] GlusterFS, https://www.gluster.org/

[2] WEIL, Sage A., et al. Ceph: A scalable, high-performance distributed file system. In: Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006. p. 307-320.

[3] LIN, W. K.; CHIU, Dah Ming; LEE, Y. B. Erasure code replication revisited. In: Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on. IEEE, 2004. p. 90-97.

[4] FIO, https://github.com/axboe/fio

[5] VERKUIL, Steven. A comparison of fault-tolerant cloud storage file systems. In: 19th Twente Student Conference on IT. 2013.

[6] What is big data?, https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html

[7] LEVY, Eliezer; SILBERSCHATZ, Abraham. Distributed file systems: Concepts and examples. ACM Computing Surveys (CSUR), 1990, 22.4: 321-374.

[8] RAO, Chung-Hwa Herman; SKARRA, Andrea H. Distributed systems with replicated files. U.S. Patent No 5,689,706, 1997.

[9] SHVACHKO, Konstantin, et al. The hadoop distributed file system. In: Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. IEEE, 2010. p. 1-10.

[10] Lustre, http://lustre.org/

# 요약

Distributed File System(DFS)은 컴퓨터 네트워크를 통해 다수의 저장장치 서버에 접근할 수 있도록 하는 파일 시스템이다. 현대의 DFS는 load balancing, location transparency, high availability, fault tolerance 등의 다양한 기능을 제공하고 있다. 그 중에서도 fault tolerance는 서버와 디스크의 failure로부터 데이터를 보호하기 위해 필요한 가장 중요한 기능 중 하나이다. 대표적인 DFS인 GlusterFS에서는 fault tolerance를 위해 데이터를 복제하여 별도의 서버에 저장하는 replcation과 데이터에 인코딩 과정을 거쳐 parity를 다른 서버에 저장하는 Erasure code를 지원하고 있다. 이 두 방법은 하나의 데이터에 대한 정보가 여러 서버 노드에 분산되어 저장되기 때문에 데이터의 정합성을 유지하기 위해서는 서버 노드 간의 data consistency 유지가 필수적이다. 만약 data consistency가 유지되지 않는다면 각 서버 노드는 다른 내용의 데이터를 저장하게 되고 이는 fault tolerance의 파괴로 이어진다. 따라서 이러한 문제를 해결하기 위해 GlusterFS에서는 매 연산을 수행할 때 모든 서버에 lock을 잡는 방법을 사용하고 있다. 이런 방법을 사용하는 이유는 파일 연산이 서버 노드 간에 '섞여서' 전달될 수 있기 때문이다. 모든 file operation은 전체 서버 노드에 반드시 atomic하게 적용되어야 한다. 하지만 현재의 GlusterFS의 구현에서는 같은 file에 대한 operation이라도 다수의 io-thread 및 event-thread에서 parallel하게 동작할 수 있기 때문에 이를 위한 concurrency control 작업이 필수적이다. 이러한 작업은 최대 2회의 추가적인 round trip이 발생할 수 있고 또 락을 관리하는 등의 오버헤드가 발생하게 된다. 따라서 우리는 같은 file에 대한 operation은 항상 같은 코어에서 수행하도록 스케쥴링하여 전체 시스템에서 같은 파일에 대한 연산의 순서를 지켜줌으로서 추가적인 concurrency control 작업 없이 서버 노드 간의 data consistency를 유지할 수 있는 방법을 제안한다. 우리는 이러한 방법을 통해 randread에서 평균 63%, 최대 83%의 성능 향상이 있었고 randwrite에서는 평균 60%, 최대 69%의 성능 향상을 얻을 수 있었다.

**주요어**: GlusterFS, Erasure Code, Cluster Lock

**학번**: 2015-21276