Performance Optimization of Communication Subsystem in Scale-out Distributed Storage

Uiseok Song, Bodon Jeong, Sungyong Park
Department of Computer Science and Engineering
Sogang University, Seoul Korea
{ussongii, vodoni20}@gmail.com, parksy@sogang.ac.kr

Kwonyong Lee
New Computing Lab, Corporate R&D Center
SK Telecom
kwonyong.lee@sk.com

***Abstract*— Scale-out distributed storage systems have recently gained high attentions with the emergence of big data and cloud computing technologies. However, these storage systems sometimes suffer from performance degradation, especially when the communication subsystem is not fully optimized. The problem becomes worse as the network bandwidth and its corresponding traffic increase. In this paper, we first conduct an extensive analysis of communication subsystem in Ceph, an object-based scale-out distributed storage system. Ceph uses asynchronous messenger framework for inter-component communication in the storage cluster. Then, we propose three major optimizations to improve the performance of Ceph messenger. These include i) deploying load balancing algorithm among worker threads based on the amount of workloads, ii) assigning multiple worker threads (we call *dual worker*) per single connection to maximize the overlapping activity among threads, and iii) using multiple connections between storage servers to maximize bandwidth usage, and thus reduce replication overhead. The experimental results show that the optimized Ceph messenger outperforms the original messenger implementation up to 40% in random writes with 4K messages. Moreover, Ceph with optimized communication subsystem shows up to 13% performance improvement as compared to original Ceph.**

Keywords—Ceph; performance optimization; distributed file system; load balancing; multi-connection;

I. INTRODUCTION

Recently, we have been experiencing an explosive growth of data being generated on the Internet and the volume of data is expected to grow further as the number of Internet of Things (IoT) devices connected to the Internet is getting bigger. The recent report by Gartner [1] predicts that the number will reach 20.4 billion by 2020. To accommodate such a vast volume of data in a more flexible and effective manner, there has been a growing interest in scale-out storage systems such as Ceph [2], Gluster [3], and Lustre [4]. Unlike scale-up systems, which scale vertically by adding more resources to the server to expand capacity, the scale-out systems can scale easily by adding more physical servers. In cases, where the amount of data increases exponentially, the scale-out storage systems are generally preferred due to their scalable, flexible and fault-tolerant characteristics.

The scale-out storage systems generally comprise of several nodes where some nodes act as storage while other as a metadata or admin nodes. These nodes continuously communicate with each other in order to accomplish the requests initiated by the clients or other nodes in the cluster. Scale-out distributed storage systems usually use check-pointing or heartbeat mechanism to detect the failure nodes by sending messages over the cluster network. Furthermore, replication is employed to ensure high data availability and reliability which also increases the network contention. All such operations incur significant communication overheads between nodes in the cluster and it will get worse as the number of nodes in the cluster increases. If the communication subsystem is not properly optimized, the overall system performance can be degraded. As the network bandwidth increases to hundreds of gigabits per second and the corresponding network traffic also increases, the communication subsystem is sometimes proven to be a performance bottleneck in distributed storage systems [5].

Among the scale-out storage systems used recently, Ceph is gaining more attention these days due to the advancement in cloud computing technologies [6]. Ceph is a distributed object-based storage system that provides various storage services such as block service, object storage service, and POSIX-based file service [2]. It is also used as a storage backend for cloud platforms due to its reliability and scalability. The *async* messenger, a default socket-based communication subsystem in Ceph, is responsible for handling various messages between client and storage servers called OSDs (Object Storage Daemons), and between OSDs. The *async* messenger has a thread pool structure [7] that creates a configurable number of worker threads in advance and manages them in a thread pool. When a new connection is established, the *async* messenger assigns it onto a worker thread from the pool in a round-robin fashion. Each worker thread uses *epoll* to check and handle incoming messages such as read or write requests. Users can configure the appropriate number of threads according to their environments.

Although the *async* messenger is widely used as a default messenger in Ceph, it has inherent performance problems. For example, the *async* messenger has a problem of balancing workloads among worker threads [8]. A worker thread is permanently allocated to a connection in a round-robin fashion without considering the amount of workload in the connection. Furthermore, each worker thread is responsible for the whole communication process from reading data from socket buffer to dispatching the data to read/write from/to the storage. This prevents other worker threads from reading data when the preceding thread is executing dispatch activity. Note that only a single worker thread is responsible for handling traffic from a single connection. In addition, Ceph uses primary copy

978-1-5090-6558-5/17 $31.00 © 2017 IEEE
DOI 10.1109/FAS-W.2017.157

replication policy for data reliability and ensures strong consistency semantic using synchronous IO [2]. Such replication operations can degrade the performance in heavy and write dominant workloads [11]. Although there have been several studies [8][9][10][11][12] that focus on either optimizing [8][9][10] or evaluating [11][12] the performance of Ceph storage system, these studies have not resolved the performance problems in communication layer aforementioned above. Among the existing research studies, the approach proposed in [8] is most similar to one of the approaches proposed in this paper, which solves the load imbalance problem among worker threads. However, the approach for overlapping different worker threads and reducing replication overheads, proposed in this paper have not been addressed.

In this paper, we have the following contributions:

- Load Balancing: We propose a simple algorithm to balance the workloads among worker threads. Our proposed algorithm uses a simple heuristic approach so that it is faster and more reactive to the workload change.
- Dual Workers and Multiple Connections: To maximize the overlapping among worker threads, we assign multiple threads to a single connection (we call *dual worker*). This allows other worker threads to read data from socket buffer while a preceding thread is on the dispatch activity. We also maintain multiple connections between OSDs to efficiently utilize the communication bandwidth, which can also reduce the replication overheads.
- Evaluation in Real Testbed: We modify the *async* messenger implementation in Ceph and evaluate the performance in real testbed. The benchmarking results indicate that the performance gain by the optimized Ceph *async* messenger is up to 40% in random writes with 4K messages when only the communication subsystems are compared, whereas, overall Ceph performance improved up to 13% when compared against the original ceph without the optimization approach.

The rest of this paper is organized as follows. Section II analyzes the problems in current Ceph *async* messenger and discusses the motivation of this paper. Section III presents three optimization approaches in detail – load balancer, dual worker, and multi-connection. Section IV compares the performance and provides performance evaluation with analysis. Section V concludes the paper.

II. Analysis and Motivation

In this section, we analyze the problems in Ceph *async* messenger and summarize our motivations using random write test with 4K block. For this, three clients generate random write traffic with the *fio* benchmark [13] and the Ceph cluster with 4 nodes, each of which has 2 OSDs (overall 8 OSDS), is used. The size of worker pool in each OSD is set to 16.

A. Overall Structure of Ceph Async Messenger

Fig. 1 depicts the overall structure of Ceph *async* messenger. The *async* messenger is one of three messengers in Ceph and is currently used as a default messenger for socket-based communication interface.

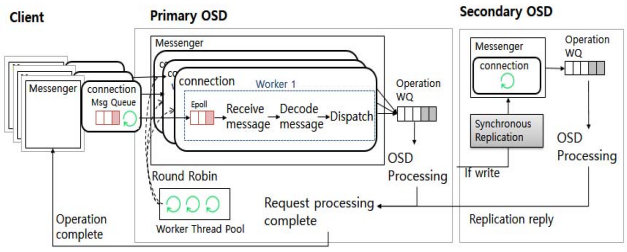

Fig. 1. Ceph message communication process using async messenger

As shown in Fig. 1, the *async* messenger initially creates a pool of worker threads in advance. The number of worker threads is a configurable parameter and can be set by users. Each worker thread in the pool continuously checks the events from clients by invoking *epoll* function. When a connection request is received, the *async* messenger assigns the connection to one of the worker threads in a round-robin fashion. While multiple connections can be assigned to a single worker thread, one connection cannot be assigned to multiple worker threads. When each worker thread receives a message from clients, it is responsible for reading the message from socket buffer, decoding the message, and dispatching the message to the OSD based on the message type. Since only one worker thread is handling messages from a single connection, the possibility of overlapping between reading and dispatching activities is excluded. If the dispatched message is a write operation, the primary OSD initiates a replication request to the secondary OSD and sends an acknowledgment message back to the client after it receives a replication completion message from the secondary OSD. This is done synchronously.

B. Load imbalance of Workers in Async Messenger

As discussed above, the *async* messenger assigns each connection to an appropriate worker thread in a round-robin fashion without considering the workload of each connection. Such connection-to-thread placement policy generates bottleneck especially in the case of heavy traffic workloads. In order to see how much imbalance each worker thread generates, we have measured the total number of events and the accumulated message size of 128 worker threads for 180 seconds by using random write operation with 4K message size. As shown in Fig. 2, the number of fetched events and their accumulated sizes are heavily imbalanced, which can harm the overall performance of Ceph storage system. For example, some threads process more than 540,000 events for 3 minutes, while there are several threads that only process less than 250 events.

Fig. 2. Number of events and total bytes processed by each worker

C. Latency Analysis in a Worker Thread

When a worker thread receives an event from *epoll* function, it carries out a series of actions such as reading the message from socket buffer, decoding and checking the message, and dispatching the message for OSD processing. This sequence is generally divided into four parts: read message, decode message, check signature and sequence number, and dispatch message. Since the message buffer is shared by the first three parts, only the last part (dispatch) can be executed independently. In order to check the possibility of overlapping between the first three parts and the last part, we have measured the latency of each part when 4K block message is received. As shown in Fig. 3, the latency between (1) and (2) is about 24 µs and the latency between (2) and (3) is about 29 µs. This result indicates that the last part can be overlapped with the first three parts if more than one thread is assigned to a single connection. However, it should be noted that if the message size gets bigger, the time taken to execute the first three parts increases linearly. Therefore, the benefit of assigning multiple threads to a single connection is minimized.

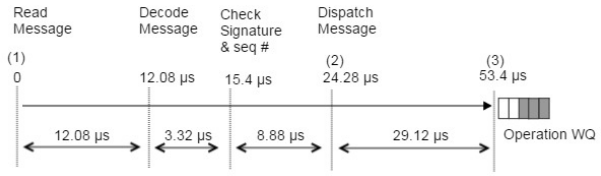

Fig. 3. Async messenger latency analysis when receiving 4K block

D. Replication Overhead and Bandwidth Analysis

As we mentioned before, Ceph uses primary copy replication policy for data reliability and supports strong consistency semantic using synchronous IO. Fig. 4 shows the sequence of a write operation when a client sends two write requests to OSD continuously. As shown in Fig. 4, the second replication operation in primary OSD is delayed until the first replication operation is sent to secondary OSDs. This means that if we have write dominant traffic, it is likely that the performance can be severely degraded. Since each write operation is accompanied by a replication operation, it is necessary to devise an efficient scheme to make replication operation finish as soon as possible.

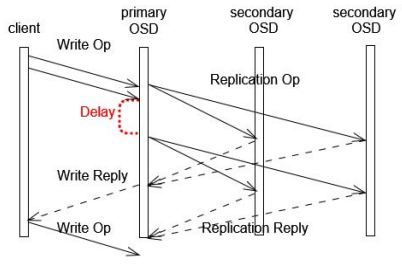

Fig. 4. Sequence of write operation with replication

In order to see the possibility of expediting the replication operation, we have analyzed how much bandwidth each connection between OSDs utilizes by varying block sizes from 4K to 2M in Fig. 5. As shown in Fig. 5, the maximum bandwidth achievable in 10Gbps network is only 5Gbps. This means that it is still possible to utilize the extra bandwidth if we create multiple connections between OSDs and transmit messages in parallel fashion at higher extent.

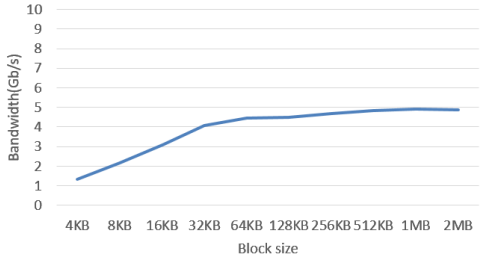

Fig. 5. Bandwidth of Ceph cluster node during random write test

III. Design and Implementation

This section presents the design and implementation of proposed optimization approaches based on the problems identified in Section II. First, we show the load balancing among worker threads. Second, we shed light on the approach assigning dual worker threads per single connection, and third, we present multi-connection approach between OSDs.

A. Load Balancing among Worker Threads

As described in Section II, the load distribution among worker threads in *async* messenger is not balanced. To solve such problem, we implement a new balancer thread in each OSD to balance the workloads. We propose a simple heuristic algorithm that is reactive to fast changes in workload. Fig. 6 shows the overall flow of the balancer thread in each OSD.

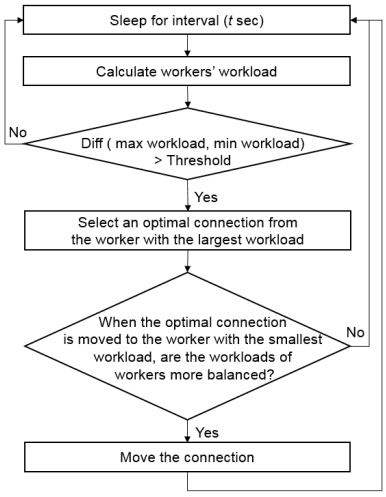

Fig. 6. Flow chart of a simple heuristic algorithm

As shown in Fig. 6, the balancer thread periodically checks the workload of every worker thread at every *t* seconds. The interval is configurable and set to 5 seconds in the proposed algorithm. In order to measure the amount of workload, we use the number of bytes processed in each worker thread as an indicator of load. When the difference between maximum workload and minimum workload exceeds a pre-defined threshold value, we trigger the load-balancing algorithm. For

the experiment, we use half of maximum workload as a threshold value. Then, the algorithm decides whether it can relocate a connection in a worker thread with the largest workload to a worker thread with the smallest workload. In order to determine a target connection, the load-balancing algorithm repeatedly checks every connection in a worker thread with the largest workload and decides an optimal connection by calculating standard deviation after relocation is complete. Finally, the load-balancing algorithm checks whether the standard deviation based on the relocation is smaller than that in the previous iteration. If it is smaller, the relocation is conducted, which ensures that unnecessary relocation is avoided. While moving the connections, we only add and delete corresponding socket descriptors using *epoll_ctl* function, so this process does not increase overall latency.

B. Assigning Dual Worker Threads per Single Connection

If clients request a new connection for data transfer, the *async* messenger assigns the connection to an appropriate worker thread that is waiting for new data using *epoll* function. Since the *async* messenger assigns one connection only to a single worker thread and the worker thread is responsible for the whole steps shown in Fig. 3, it is impossible for the worker thread to overlap independent activities within the four steps to improve performance. Therefore, we modified the *async* messenger so that multiple worker threads (we call *dual worker*) can handle traffic from the same connection, as shown in Fig. 7. To implement this, we maintain a new pool of threads just for dual workers and dynamically assign them to the same connection only if the connection handles specific messages (e.g., *CEPH_MSG_OSD_OP* and *MSG_OSD_REPOP* messages) that contain data to be stored. Although the overhead of lock contention to share common data structures between dual worker threads can increase, this approach is beneficial especially when the traffic between clients and OSDs is heavy.

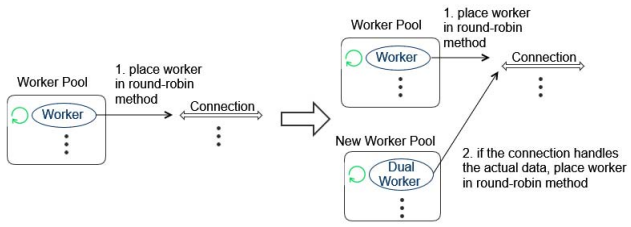

Fig. 7. Dual worker threads in a single connection

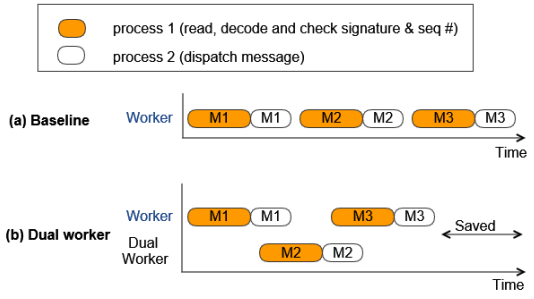

Fig. 8. Benefits of dual worker threads in a single connection

Among the four steps discussed in Fig. 3, the dispatch activity in a worker thread is independent of other three activities. As shown in Fig. 8(a), if only one worker thread handles traffic from a single connection, the thread processes all events from the connection sequentially. On the other hand, if dual worker threads handle the events simultaneously, we can reduce the total time taken as shown in Fig. 8(b). It is worthy to note that as the size of message increases, the latency of read process accounts for almost 90% of the total latency, which reduces the benefits of overlapping. This means that the dual worker approach is more beneficial when we transfer small-sized message blocks.

C. Using Multiple Connections between OSDs

Using multiple connections for file transfer has been successful to improve the performance of communication subsystem due to the parallel execution of multiple threads [14][15]. Since the write and replication operation in Ceph are highly synchronous, it is likely that the data transfer between OSDs over multiple connections improves throughput and thus reduces the replication overheads.

In current Ceph, the *async* messenger creates only one connection between components for sending or receiving messages. We modified the *async* messenger so that it creates two connections only between OSDs (i.e., between cluster messengers) as shown in Fig. 9. The two created connections are bundled and treated like a single connection, thus share variables such as lock and message sequence. This causes additional lock overhead. After the messenger creates two connections, it uses them to transfer messages between OSDs. When the messenger transmits messages over the two connections, it alternates between two connections in a round-robin fashion. On the other hand, when the messenger receives messages, an appropriate worker thread assigned to either connection is responsible for receiving the messages. This means that it is possible for two subsequent messages not to arrive in correct order. Therefore, if the messenger receives an out-of-order message, it leaves the message in a sequence queue and delivers the message later once the message with correct sequence number arrives.

We mentioned in Section II (Fig. 4) that when a client sends two write operations continuously to the primary OSD over a single connection, the second replication operation in primary OSD should wait until we finish sending the first replication operation to secondary OSDs. However, if we apply the multi-connection approach to eliminate this bottleneck, the primary OSD can send replication operation to the secondary OSDs simultaneously. This results in more efficient bandwidth utilization and overhead reduction of replication operations.

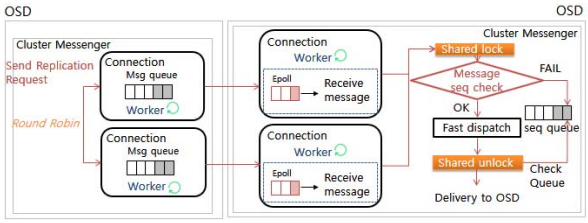

Fig. 9. Using multiple connections between OSDs

IV. EVALUATION

This section compares the performance of optimized *async* messenger with that of original messenger in Ceph version 10.2.3 and shows the impact of three optimization techniques on the performance. To achieve this, we have implemented different versions of optimized *async* messenger that includes different optimization techniques: *async-ld* (optimized *async* messenger with load balancer), *async-dw* (optimized *async* messenger with dual worker), *async-mc* (optimized *async* messenger with multi-connection) and *async-mc-ld-dw* (optimized *async* messenger combining all three optimization techniques). The *async-ld-GA* is the *async* messenger with load balancing based on GA proposed in [8].

We evaluate the performance with two different settings: messenger only test (*Msgr test*) and full Ceph test (*Full test*). In order to compare with the former setting, we also modified the Ceph code so that the messenger returns its control as soon as each worker thread dispatches events to the OSD that is responsible for placement group (PG) processing, journaling, and FileStore (or BlueStore). For the comparison, we used IOPS (Input/Output Operations per Second) with two random workloads: random read and random write.

A. Experimental Setup

The testbed used to evaluate the Ceph consists of three client nodes and a Ceph cluster with four nodes, where each node runs CentOS 7.3.1611 (kernel version 3.10.0-514) and is equipped with 10 cores, 32 GB RAM, and 2 SSDs. Each node installs Ceph version 10.2.3 with two OSDs per node (one OSD per SSD). We use a *fio* benchmark [13] with *librbd* support to vary block size and various other parameters to generate different traffic types. The number of replication and the size of worker thread pool are set to 2 and 16, respectively.

B. Performance Evaluation

We performed a random read/write test with various block sizes of 4 KB, 8 KB, 64 KB, and 1 MB. Each client generates random read/write traffic with the *iodepth* and *numjobs* parameters in *fio* benchmark set to 4 and 8, respectively. Figs. 10 and 12 show the comparison of IOPS for random write and read operations with two different settings (*Msgr test* and *Full test*).

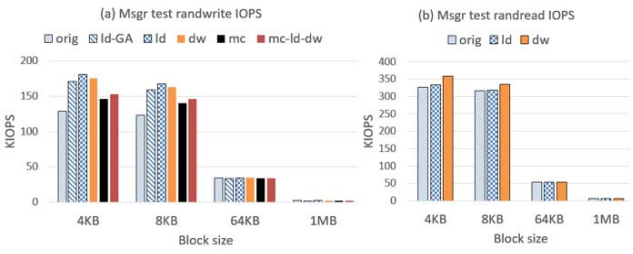

Fig. 10. Comparison of random write/read performance in *Msgr test*

1) Comparison of Random Write/Read in Msgr test : As shown in Fig. 10(a), all optimized *async* messengers outperform the original messenger in random write benchmark with small-sized blocks such as 4KB and 8KB, and *async-ld* performs the best among them. For example, *async-ld* outperforms *async-orig* by 40% and 35% for 4 KB and 8 KB, respectively. *async-ld* also shows better performance than *async-ld-GA* by about 6%. However, we observe that the performance gain is disappeared as we increase block size. From a careful trace of system data, we found that the number of dropped TCP packets was getting larger as we increase block size. Thus, we cannot fully utilize the benefits of three optimization techniques in large block sizes. Using other congestion control algorithms is another option, which needs to be investigated later.

Surprisingly, the *async* messenger even with all optimization techniques included (*async-mc-ld-dw*) does not perform well compared to *async-ld* or *async-dw*. Note that we share a single thread pool to implement the multi-connection approach. This means that if increase the number of OSDs which require multiple connections, the available worker threads in the pool becomes decreased. In the dual worker approach, the number of threads running in the system is doubled since we maintain a separate thread pool for dual workers. Therefore, we estimate that there is a strong relationship among thread scheduling overheads by increasing the number of threads, workload pattern, and the hardware capacity (e.g., the number of cores and CPUs, memory size, network speed, etc.).

Fig. 10(b) compares the performance of random read by varying different block sizes. Since multi-connection mainly targets at reducing replication overheads incurred by write operations, we exclude the performance results using multiple connections in this benchmark. As a result, *async-dw* shows the best performance and outperforms *async-orig* by 9% and 5% for 4 KB and 8 KB block sizes, respectively. On the other hand, the performance improvement using *async-ld* is minor and we cannot get any improvements as we increase block size. This is because, in the load-balancing algorithm, we define load as the size of all processed messages. Considering that the read operation does not contain any data to process, the effect of using the load-balancing algorithm in read operations should be minimal. In large block sizes, *async-dw* does not perform well since the latency of read process is likely to take up a large portion of the total latency as we increase block size.

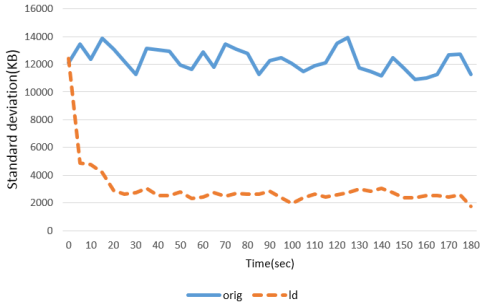

Fig. 11. Standard deviation of random write test using 4K block

2) Effect of load balancing in Msgr test : In order to see how well *async-ld* balances the workloads in write operations, we have traced the standard deviations of the size of messages processed by each worker in intervals of 5 seconds. As shown in Fig. 11, the standard deviation for every 5 seconds does not show any significant changes between about 10,000KB and

14,000KB in *async-orig*. However, the *async-ld* shows a significant drop of standard deviation from 12,000KB to 2,000KB, which indicates that the load balancer balances workloads better than *async-orig*.

3) Comparison of Random Write/Read in Full test : Fig. 12(a) and (b) illustrate the performance of random write and read operations in *Full test*. As expected, *async-mc-ld-dw* outperforms *async-orig* by 13% and 10% for small-sized blocks. However, unlike the results in *Msgr test*, the performance improvement becomes smaller and *async-mc-ld-dw* outperforms other competitors. In the random read test, we cannot observe any improvements. Although these discrepancies arise from many different factors, we suspect that the activities related to journaling, PG processing, and FileStore (or BlueStore) influence the overall performance. Our preliminary experimental results showed that the performance degraded highly when the incoming messages pass through the PG processing step. We are investigating in detail to find out the optimal configuration parameters to evaluate such cases.

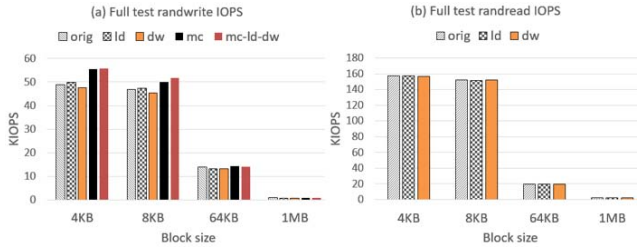

Fig. 12. Comparison of random write/read performance in *Full test*

4) Multiple Connections for Bandwidth Improvement in Full test : Fig. 13 shows the bandwidth improvement rate for *async-mc* and *async-mc-ld-wd* against *async-orig* in *Full test*. Similar to the results shown in Fig. 12(a), the bandwidth also increases by up to 17%. Note that our multi-connection approach increased bandwidth, which led to the increased throughput.

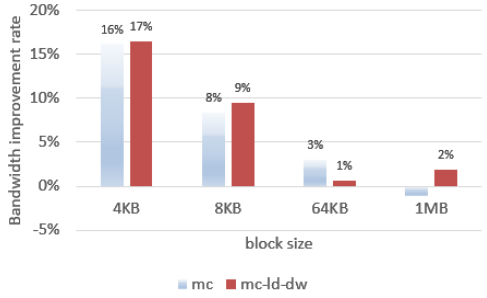

Fig. 13. Bandwidth improvement rate against *async-orig* (random write)

V. CONCLUSION

Scale-out distributed storage system, Ceph, is being deployed with high-speed network connectivity for communication between nodes in the cluster. However, Ceph does not fully utilize available bandwidth due to poor design of communication subsystems. In this paper, we spotlight the importance of such communication subsystem and propose three techniques mainly, i) load balancing, ii) overlapping receive and message dispatch operation, and iii) using multi-connections to improve performance of communication subsystems. To show the feasibility of proposed ideas, we implemented these ideas in Ceph. Our approach showed 40% improvement when only messenger-messenger performance is compared and overall system performance improved 13% when compared to original Ceph with no optimized communication subsystem.

A preliminary analysis indicated that the PG processing is one of the bottlenecks that hide most of the performance gain from the optimized messenger. We need to answer detailed reasons for those issues after a more in-depth analysis. Further analysis with real workloads is also necessary to prove our approach in general.

ACKNOWLEDGMENT

This work was supported by SK Telecom in Korea and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.B0101-16-0644, Research on High Performance and Scalable Manycore Operating System).

REFERENCES

[1] Gartner, http://www.gartner.com/newsroom/id/3598917

[2] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System", 7th Symposium on Operating Systems Design and Implementation. USENIX Association, 2006, pp. 307-320

[3] Gluster, https://www.gluster.org/

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

[5] Izumobase Blog, "Inline Deduplication for Distributed Storage", http://www.izumobase.com/blog/inline-deduplication-for-speed

[6] Openstack User Survey, 2015. http://www.openstack.org/assets/survey/Public-User-Survey-Report.pdf

[7] Y. Ling, T. Mullen, and X. Lin, "Analysis of Optimal Thread Pool Size", ACM SIGOPS Opearating Systems Review, Vol 34, Issue 2, April, 2000, pp. 42-55

[8] Y. Han, K. Lee, and S. Park "A Dynamic Message-aware Commuication Scheduler for Ceph Storage System", AMGCC, 2016

[9] K. Zhan and A. Piao, "Optimizaiton of Ceph Reads/Writes based on Multi-threaded Algorithm", HPCC-SmartCity-DSS, 2016

[10] M. Oh, J. Eom, J. Yoon, J. Yun, S. Kim, and H. Yeom ,"Performance Optimization for All Flash Scale-out storage", IEEE international Conference on Cluster Computing, 2016, pp. 1561-1563

[11] D. Gudu, M. Hardt, and A. Streit, "Evaluating the Performance and Scalability of the Ceph Distributed Storage System", IEEE International Conference on Big Data, 2014

[12] D. Lee, K. Jeong, S. Han, J. Kim, J. Hwang, and S. Cho, "Understanding Write Behavior of Storage Backends in Ceph Object Store", MSST, 2017

[13] Fio benchmark, http://freecode.com/projects/fio

[14] T. Ito, H. Ohsaki, and M. Imase, "GridFTP-APT: Automatic Parallelism Tuning Mechanism for Data Transfer Protocol GridFTP", 6th IEEE Interntional Symposium on Cluster Computing and the Grid (CCGrid), 2006.

[15] H. Subramoni, P. Lai, R. Kettimuthu, and D. K. Panda, "High Performance Data Transfer in Grid Environemnt Using GridFTP over InfiniBand", Technical Report, OSU-CISRC-11/09-TR53