A Dynamic Message-aware Communication Scheduler for Ceph Storage System

Yunjung Han, Sungyong Park
Department of Computer Science and Engineering
Sogang University
Seoul, Republic of Korea
hanmar88@naver.com, parksy@sogang.ac.kr

Kwonyong Lee
SDS Tech. Lab, Corporate R&D Center
SK Telecom
Seoul, Republic of Korea
kwonyong82@gmail.com

***Abstract* — With the proliferation of cloud computing technologies, the Ceph, a distributed object-based storage system has been an attractive alternative to building a storage backend due to its excellent performance, reliability, and scalability. As the storage system processes huge amount data and the network traffic generated from the cloud increases rapidly, designing a high-performance messenger in the storage system has created a lot of challenging issues. Although the *async* messenger, one of the Ceph's messengers, is known to be efficient and flexible, it contains several performance problems due to its simple round-robin based scheduling scheme that assigns a connection to a worker thread without any consideration for the amount of workloads transferred through the connections. This causes the imbalance of worker threads and adversely affects the performance of the Ceph storage system. This paper proposes a dynamic message-aware communication scheduler for Ceph storage system that balances the workloads of worker threads based on the types of incoming messages, while avoiding unnecessary connection movements among worker threads. We use genetic algorithm (GA) to solve this problem and implement the proposed scheduling algorithm using the *async* messenger. The benchmarking results show that the proposed approach outperforms the original *async* messenger by as much as 12.5% under the same workload from clients and 24% under the random workloads from clients.**

Keywords—distributed object storage system; Ceph; async messenger; message-aware; load balancing; thread scheduling

I. INTRODUCTION

Recent developments in technology have led to the massive growth of data on the Internet. There is a growing trend to store and manage massive-scale data in clouds to make data management more flexible and effective. As the demands on clouds increase, using a high performance distributed storage as cloud backend is becoming more crucial. Typical distributed storage systems include Ceph [1], Gluster [2], and Lustre [3]. Among them, Ceph is an open-source project and a massively scalable and fault-tolerant distributed file system, which is also popular in cloud storage solutions. For example, Openstack [4], one of the most well-known open-source cloud platforms, uses Ceph in its service modules [5].

In a distributed storage system such as Ceph, there is an important component called communication messenger that takes in charge of sending or receiving various messages from the clients or among the cluster members. As the bandwidth of network hardware increases to hundreds of gigabits per second and the network traffic from the cloud also increases [6], the communication messenger is sometimes proven to be a performance bottleneck in distributed storage systems.

Ceph has three types of messengers: *simple*, *async*, and *xio*. The *simple* messenger has a literally simple structure called thread per connection [7], where a thread is generated whenever a new connection is created. If a Ceph cluster consists of many physical nodes and has many clients sending requests, plenty of connections can be created to communicate with other physical nodes or processes. These include object storage daemons (OSDs) and monitors to process requests from other Ceph components. Especially in a limited environment where a physical node has a small number of cores or insufficient memory, it is possible that running too many threads may cause performance degradation due to frequent context switching and memory exhaustion. In contrast, the *async* messenger has a thread pool structure [7] that creates a configurable number of threads in advance and manages them in a thread pool. Users can configure appropriate number of threads according to their environment**.** It is generally reported [8] that the *async* messenger outperforms the *simple* messenger, which is also verified by the performance evaluation in our environment. The *xio* is a messenger specially designed for the Infiniband-based network environments.

The *async* messenger has a problem of balancing the workloads among worker threads in a worker pool (worker thread: a thread 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. In addition, once a connection is assigned to a particular worker thread, this mapping is not changed until the connection is finished, even if the connection traffic varies considerably over time. Since the *async* messenger is supposed to handle various message types with different priorities, sizes, and process times, a certain number of worker threads can be overloaded while other threads are idle (or process only a few messages). For example, the number of requests from heartbeat connections is only a few, and they are small-sized and have low priorities. Meanwhile, the connections for replication or client requests such as disk read or write include large and high-priority messages. This imbalance can negatively influence on the messenger performance and Ceph overall.

There have been several studies [9–11] that focused on scheduling techniques related to data transfer in a distributed storage system. Sinbad [9] is a system that identifies imbalance and selects replica destinations to prevent congested links. This

978-1-5090-3651-6/16 $31.00 © 2016 IEEE
DOI 10.1109/FAS-W.2016.25

shortens the completion time of data transfer. The system focuses on relieving network link congestion from the sender's point of view by choosing a data transfer target. Also, Sinbad is a master–slave structure, which is not applicable to distributed storage systems without a centralized component such as Ceph. SEAL [10] reduces the average slowdown and turnaround times by varying the concurrency of individual transfers through considering the resource limits of the target node while avoiding saturation from the sender's point of view like Sinbad. Unlike Sinbad and SEAL, we concentrate on workloads generated by multiple senders from the receiver's point of view, and schedule the threads to balance their workloads. LADS [11] optimizes data transfers in a distributed file system. LADS focuses on optimizing the end-to-end transfer of bulk data and schedules data by exploiting the underlying storage layout, but we concentrate on scheduling communication threads at endpoints by using the message characteristics, such as the type, size, and priority.

In this paper, we propose a dynamic message-aware communication scheduler for Ceph file system to solve the problem of workers' imbalance, which results in improving the performance. The proposed scheduling algorithm balances the workloads of worker threads based on the types of incoming messages, while avoiding unnecessary connection movements among worker threads. For example, low priority messages such as messages from heartbeat connections are assigned to a particular thread in order not to interfere with other high priority messages. On the other hand, high priority messages are evenly distributed to each worker thread to balance the workloads among threads. We use genetic algorithm (GA) to formulate this problem. By incorporating the connection movement rate into the fitness function, the proposed algorithm also minimizes unnecessary connection movements. We have implemented the proposed scheduling algorithm using the *async* messenger and compared its performance with the original *async* messenger. The benchmarking results show that the proposed approach outperforms the original messenger by as much as 12.5% under the same workload from clients and 24% under the random workloads from clients.

The rest of this paper is organized as follows. Section II presents the motivation of the work proposed in this paper. Section III briefly explains previous studies. Section IV defines and formulates the problem. Section V discusses our message-aware scheduling algorithm and the performance of our scheduling techniques is given in Section VI. Section VII concludes this paper.

II. Motivation

A. *Structure of an async messenger in OSD*

Ceph's components such as OSDs and monitors have the same messenger object: *simple*, *async*, or *xio* messenger. Therefore, we focused on messengers in OSDs because they handle practical data in Ceph. An OSD is responsible for storing objects on a local file system and providing access to them over the network.

An OSD process has different types of messengers. There are six types of *async* messengers: client, cluster, hbclient, hb_back_server, hb_front_server, and ms_objector. Each messenger deals with different message types, as listed in Table I. These statistics were measured during a *randread* and *randwrite* test from the Fio benchmark [12] for about 180s. Messengers such as hbclient, hb_back_server, and hb_front_server process messages related to the heartbeat operations among the nodes or processes such as OSDs and monitors. The cluster messenger is in charge of processing messages for replication operations, and the client messenger handles requests sent from external clients. Table I also shows the number and priority of the messages with various message types. For example, CEPH_MSG_OSD_OP, MSG_OSD_REP_OP, and MSG_OSD_REPOPREPLY have higher priorities than other message types. It is noticeable that the number of messages received by each messenger differs significantly. For example, when the client messenger is connected to client nodes, it receives many messages if clients send bulk messages. On the other hand, when the hb_back_server messenger creates a connection to communicate with other OSDs, it receives relatively few messages, and these are small and low in priority. Thus, there are large differences in workloads among connections. However, the *async* messenger does not consider these message characteristics at all when assigning a connection to a worker thread.

TABLE I. MESSAGES PROCESSED BY EACH MESSENGER

Messenger Type	**Message's Characteristics**			
	Message Types	***priority***	***The number of messages in randread***	***The number of messages in randwrite***
client	CEPH_MSG_OSD_OP	high	20404227	5201621
cluster	CEPH_MSG_OSD_MAP	low	92	123
	MSG_OSD_PG_NOTIFY	low	2220	2359
	MSG_OSD_PG_QUERY	low	1033	1128
	MSG_OSD_PG_LOG	low	386	440
	MSG_OSD_PG_INFO	low	3677	3874
	MSG_OSD_REPOP	high	0	5641431
	MSG_OSD_REPOPREPLY	high	0	6650456
hbclient	MSG_OSD_PING	low	28724	28724
hb_back_server			14356	14356
hb_front_server			14356	14356
ms_objector	-	-	-	-

A process such as an OSD or monitor actually manages all types of *async* messengers in a worker pool with configurable number of worker threads. If a new connection is created in one of the messengers, it is mapped to one of the worker threads in the worker pool. If the number of connections is greater than the number of worker threads, it is possible that several workers manage more than one connection. When a connection is assigned to a worker thread, the socket description of the connection is registered to the worker's event center provided by event multiplexing techniques such as *epoll*, *kqueue*, or *select*. In this paper, we use *epoll*. When a message arrives at a connection, the worker thread assigned to the connection can detect the event by using *epoll_wait*. If a worker thread has more than one connection, it has multiple file descriptions registered for its event driver. This means that a worker thread with multiple connections receives messages through them consecutively.

Fig. 1 describes how a connection is assigned to a worker thread on a first-come first-serve basis. Worker threads are scheduled according to a round-robin algorithm. With round-robin assignment, certain worker threads may be overloaded to handle multiple connections while other workers receive few requests. When the connections that usually process many high-priority messages are assigned to only a few worker threads, this can cause performance degradation.

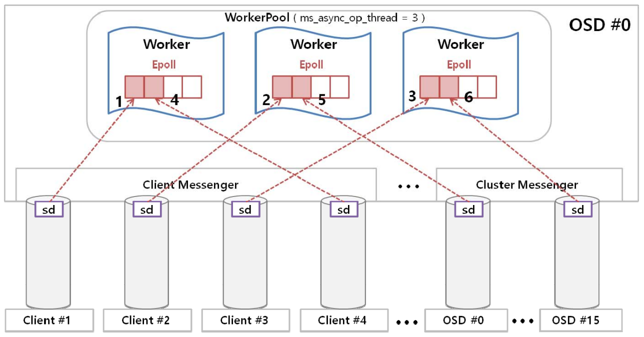

Fig. 1. Placement rule of workers and connections in *async* messenger

B. *Load imbalance of workers in async messenger*

Fig. 2 depicts the number of received messages of a worker thread in Ceph OSDs. The data were measured from a *randwrite* test from the Fio benchmark. When the test was performed, the Ceph cluster had four physical nodes and 16 OSDs, so each node had four OSDs. There were eight worker threads, so the 16 OSDs had 128 workers in total. The figure indicates that the standard deviation of the messages received by the 128 workers was very large. The largest number of messages received by a worker was 846,532, but the smallest number was only 594. As expected, certain worker threads received many messages, and some worker threads processed few messages. This situation can adversely affect Ceph's performance.

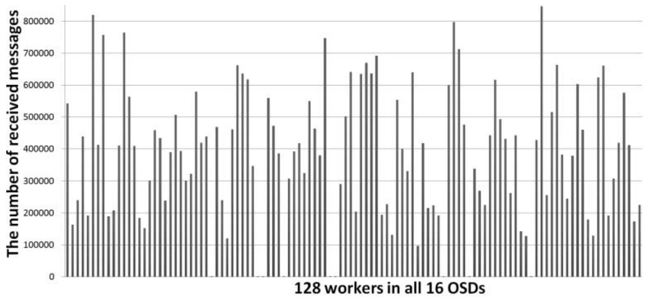

Fig. 2. The amount of messages by each worker in all OSDs.

To solve this problem, we propose a dynamic message-aware communication scheduler for Ceph storage system. From this situation, we have two goals: (i) improve the performance of Ceph messenger, (ii) reduce the variation of performance and balance processing requests among clients.

III. Related Works

Sinbad [9] tries to reduce the completion time of data transfer by selecting endpoints for replication to prevent network congestion, but it does not consider stages after data arrive. Unlike Sinbad, we focus on the receiving phase in network flow by scheduling threads that are in charge of receiving messages from socket. This technique is only valid when the master in a master–slave structure is alive, so it is not suitable for decentralized storage systems such as Ceph. SEAL [10] reduces the average slowdown and turnaround times by varying the concurrency of individual transfers through considering the resource limits of the target node from the sender's point of view. It does not consider the concurrency of receiving requests. In contrast, we focus on the concurrency of receiving requests from a socket to schedule threads from the receiver's point of view. This improves the performance and allows requests to be processed in a balanced manner. LADS [11] is a bulk data transfer framework that reflects the underlying storage layout at each endpoint to minimize I/O contention at the data source and sink. It focuses on optimizing end-to-end data transfer by using the underlying storage layout at each endpoint without negatively impacting the performance of shared storage resources for other users. In contrast, we focus on scheduling communication threads at endpoints by using message characteristics such as the size, process time, and priority to improve the performance of Ceph and by serving client requests fairly.

IV. Problem Definition

Given a set of *m* connections and a set of *n* workers, the problem is to select *n* disjoint subsets of connections from *m* so that the workload of each worker is well balanced (i.e., minimize the standard deviation of workload in each worker thread as shown in (1)), while minimizing unnecessary connection movements among worker threads. This problem can be formulated as follows:

$$\sigma = \sqrt{\frac{1}{n_w}\sum_{j=1}^{n_w}(l_{w_j} - l_{avg_w})^2}, \tag{1}$$

$$\text{s.t.} \quad l_{w_j} = \sum_{i=1}^{n_{con}} l_{con_i} \cdot x_{ij}\,, \quad i = 1,2,\cdots,m, \tag{2}$$

$$x_{ij} \in \{0,1\}, \quad j = 1,2,\cdots,n, \tag{3}$$

$$l_{avg_w} = \frac{1}{n_w}\sum_{j=1}^{n_w} l_{w_j} \tag{4}$$

$$\text{maximize} \quad \textit{fitness} = \frac{1}{\sigma + \sigma * \alpha * \frac{n_{mov}}{n_{con}}} \tag{5}$$

Here, n_w is the number of workers (i.e., *m*) and n_{con} is the number of connections (i.e., *n*). l_{w_j} is the workload of the *j*th worker, which is the sum of workloads of connections assigned to the worker and l_{avg_w} is the average of workloads from all workers calculated by (4). In (2), l_{con_i} is the workload of a connection *i*, which is currently defined by the sum of message sizes received through the connection. We assume that the bigger the message size is, the more resources (such as memory or CPU) the worker consumes. In other words, it takes more time for the worker to process large messages than small messages. Also, if the connection *i* is assigned to the worker *j*, x_{ij} is 1; otherwise, it is 0.

On the other hand, it is also more efficient not to relocate connections among worker threads as much as we can. Therefore, we also consider the connection movements in order to find the best placement. To satisfy this, we define a fitness value that combines the standard deviation of workload in each worker thread with connection movement rate as shown in (5).

In (5), σ means the standard deviation and n_{mov} is the number of connection movements. α is a constant value which represents the weight of connection movement rate in regards to σ. We used 0.1 as the value of α in this paper. Therefore, the objective of the proposed algorithm is to maximize the fitness value defined in (5).

V. DYNAMIC MESSEGE-AWARE SCHEDULING ALGORITHM

To solve the problem incurred by the imbalance of workers, we have implemented a dynamic message-aware communication scheduler using *async* messenger for the Ceph storage system. This section describes the details of how the proposed communication scheduler schedules workers in two phases. The first phase is to map a heartbeat's connection to a worker and the second phase is to balance the workloads of workers while minimizing unnecessary connection movements among worker threads. We solve it by using *Genetic Algorithm* (GA).

A. Mapping a heartbeat connection to a worker

The first phase is the *static* mapping phase of a *heartbeat connection* to a worker when a Ceph cluster is initially configured. The connections for *heartbeats* are created in hbclient, hb_back_server, and hb_front_server messenger and they mostly transmit MSG_OSD_PING type of messages. Using the traces given in Chapter II, we have noticed that *heartbeat* messages are exchanged within clusters at every configurable interval, usually 5 sec**.** Therefore, the network traffics from heartbeat connections are uniform in a sense that the traffic generated from one heartbeat connection is almost the same as that of other heartbeat connections. However, if the scale of Ceph becomes bigger, the number of heartbeat connections becomes larger. For example, an OSD creates 52 connections in our environment where the Ceph cluster has four physical nodes and each node has four OSDs. If 52 connections are involved, the complexity of the scheduler becomes higher. Therefore, it is necessary for these uniformed connections to be assigned to several workers in a balanced manner. In order to map heartbeat connections to several workers in the balanced manner, we use a round-robin algorithm for three heartbeat-related messengers.

B. Balancing workloads of workers using Genetic Algorithm

The second phase is to balance the workloads among worker threads using a meta-heuristic method, *genetic algorithm* (GA). In order to apply GA to the problem, we define a chromosome, as shown in Fig. 3. The chromosome is a linked list containing information about the assignment of connections to worker threads. For example, the first item of the chromosome in Fig. 3 has number 4, which means that the *1st* connection is assigned to the *4th* worker thread and thus the value of $x_{1,4}$ is 1. In particular, the connections are added to the chromosome list when they transfer high-priority messages or they have bursty traffic. By using the traces discussed in Chapter II, we have found that the connections containing specific types of messages such as CEPH_MSG_OSD_OP, MSG_OSD_REPOP, and MSG_OSD_REPOPREPLY usually generate bursty traffic. CEPH_MSG_OSD_OP is the message type of requests by external client, so its traffic varies by the amount of client's requests. MSG_OSD_REPOP and MSG_OSD_REPOPREPLY are the replication-related types. The connections transferring such types of messages have various traffic patterns over times because the place to replicate varies according to each request. For this reason, the connections transferring those messages are registered in the chromosome list. If connections do not contain any of the three message types, they are not registered in the chromosome list. In other words, we only consider active and bursty connections to balance the worker threads.

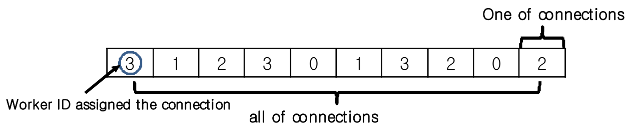

Fig. 3. An example of a chromosome

Our scheduling algorithm using GA consists of the following four steps.

1) Create the first generation: The first chronosome is current assignment and the rest of genes are created by randomly selecting the worker ids in charge of each connection. We generate 8 chromosomes in total.

2) Selection: The selection phase of GA is to select chronosomes to be used in the next generation. We select two solutions from all genes by using the "roulette wheel" method, which is likely to select the better solution with a higher probability. We calculate the fitness value of each gene that combines the standard deviation of workload in each worker thread with connection movement rate as shown in (5).

3) Crossover: One point crossover is used in this paper. It is performed by selecting one random number (larger than 0 and less than n_{con}) as a split point, spliting two parents at this point, and creating two new children by exchanging the tail part of each parent.

4) Mutation: Mutation is implemented by randomly resetting the assigned worker id. One of the connections is randomly chosen and assigned to the worker randomly. The mutation rate, which should be small, is 0.05. Then, we go back to the selection phase and repeat from step 2) to step 4) 50 times to get the best solution.

In addition, the GA is performed periodically because some connections have inconsistent traffic over times. We monitor the workloads of worker threads at every *t* interval. If the interval *t* is too small, an overhead to calculate the algorithm occurs. If the interval *t* is too large, there is little effect on the scheduling. The algorithm also checks whether any newly created connections exist and the fitness value exceeds a threshold δ at every interval *t*. We use 5 sec as interval *t*.

VI. PERFORMANCE EVELUATION

A. Experimental Environment

As shown in Fig. 4, the Ceph cluster consists of 4 nodes, where each node in the cluster is equipped with 48 cores and 256GB RAM. Each Ceph cluster node is configured to have 4 OSDs and each OSD uses 2 SSDs configured with RAID 0. The 8 Fio clients are used to generate the workloads for the experiments. Both Ceph cluster nodes and clients are all connected with 10 Gbps Ethernet. The CentOS 7.1.1503 with kernel version 3.10.0-123 and the Ceph version 0.94.1 are used to conduct the experiments and the *async* messenger version 9.0.2 is modified to include the proposed scheduler. We use Fio benchmark [12] with krbd (kernel rados block device) support to evaluate the performance of *async-orig* and *async-GA*. The *Async-orig* is the original version of *async* messenger and the *async-GA* is a modified *async* messenger that includes the GA-based scheduler to balance the workloads among workers. In order to focus only on the communication part of Ceph storage system, we slightly modified the original Ceph such that the messenger returns as soon as it finishes its job.

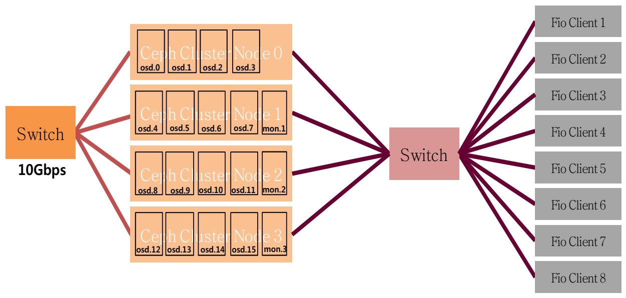

Fig. 4. Testbed for Ceph cluster

B. *Performance comparison*

We measured the performance using two types of workloads from 8 Fio clients. The 1st type was to generate the same workload from each client, while the 2nd type was to generate the random workload from each client. We have conducted several experiments with three operations such as *randread*, *randwrite*, *randrw* with 4K messages and the ratio of *randread* to *randwrite* in *randrw* operation was 8:2. The Fio parameters such as *iodepth* and *numjobs* were all set to 8 in all clients.

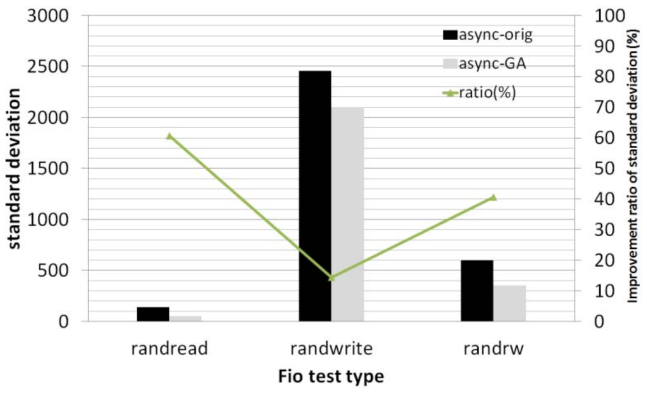

Fig. 5. Comparison of standard deviations using the same workloads

Fig. 5 compares the standard deviations among worker threads to check how well the workloads are balanced. As shown in Fig. 5, the standard deviations of *async-GA* using three operations are lower by 60%, 14%, and 40% than those of *async-orig*. This indicates that the message-aware scheduling used in *async-GA* balances the workloads better than that of *async-orig*, which just balances the connections in a round-robin fashion without considering the real workloads inside the connections.

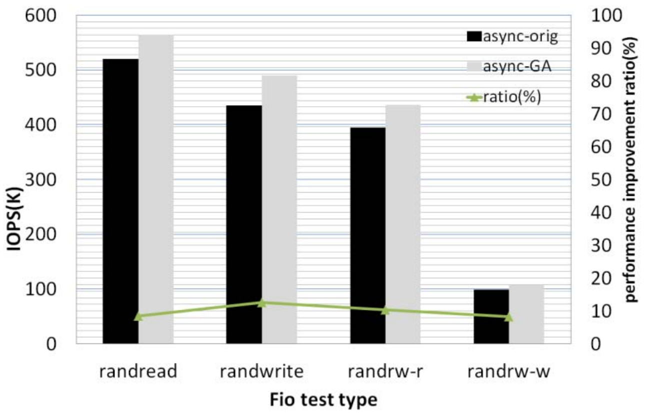

Fig. 6. Comparison of IOPS using the same workloads

Fig. 6 compares the IOPS values for *async-orig* and *async-GA* using the same operations in order to see whether the load balancing may result in improving overall I/O performance or not. The IOPS values for *randrw-r* and *randrw-w* represent the read IOPS for *randrw* operation and the write IOPS for *randrw* operation, respectively. As we have expected, the *async-GA* outperformed the *async-orig* by 8%, 12%, 10%, and 8% for all three operations.

Since it is generally inappropriate to clearly see the effectiveness of load balancing algorithm with the experiments using the same workloads, we decided to repeat the same experiments with random workloads. In this experiment, the Fio parameters such as *numjobs* and *iodepth* in each client were set randomly and thus the workloads from clients were randomly generated using normal distribution.

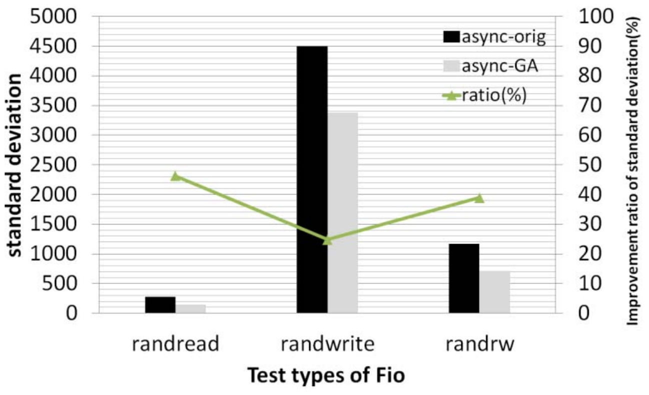

Fig. 7. Comparison of standard deviations using the random workloads

Fig. 7 compares the standard deviations among worker threads using the random workloads. Like the results obtained using the same workloads (Fig. 5), the standard deviations of *async-GA* are again lower by 46%, 24%, and 38% than those of *async-orig*. It is worthy to note that our GA-based algorithm well balances the workloads regardless of the workload types.

Fig. 8 compares the IOPS values for *async-orig* and *async-GA* using the random workloads to figure out the relationship between the IOPS values and the workload type. As shown in Fig. 8, the *async-GA* again outperformed the *async-orig* by 17%, 24%, 20%, and 19% for three operations. However, the performance improvement using the random workloads is bigger than that using the same workloads. Considering that the random workloads are more likely to deliver uneven workloads to worker threads, we can expect that the performance gain using the proposed algorithm becomes bigger as the degree of randomness in workloads increases.

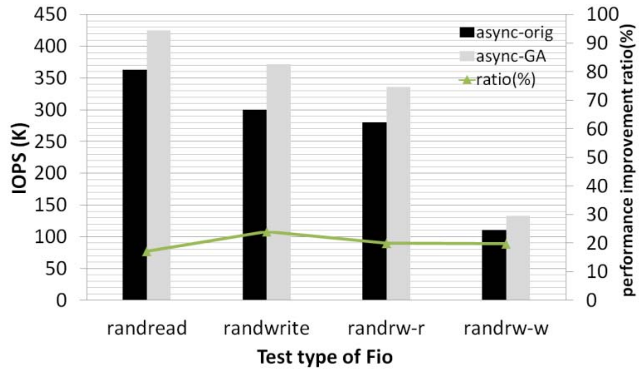

Fig. 8. Comparison of IOPS using the random workloads

VII. Conclusion

In this paper, we have proposed a dynamic message-aware communication scheduler that balances the workloads among worker threads based on the type of messages transferred through the connections. We have also considered the connection movement rate in order to avoid unnecessary connection movements, which may degrade the performance if the connections are frequently migrated to other threads during the execution. We formulated the problem with GA and implemented the proposed algorithm using the original *async* messenger. The benchmarking results showed that the proposed algorithm outperformed the original implementation in both the same and the random workloads.

Although the benchmarking results were positive, there are several points to improve our current implementation. For example, current algorithm assumes that there are fixed number of workers. However, it is more flexible if the number of workers can be configured dynamically as the amount of workloads increases over times. Also, it is possible that the prediction of workload pattern may improve the algorithm further. We are currently investigating those issues.

Acknowledgment

This research was supported by SK Telecom in Korea.

References

[1] Sage A. Well, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn, “Ceph: A scalable, high-performance distributed file system,” Proceedings of the 7th symposium on Operating systems design and implementation, pp. 307–320, November 2006.

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

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

[4] Openstack, https://www.openstack.org/.

[5] Sebastien Han, “Back from the summit: Ceph/OpenStack integration,” http://techs.enovance.com/6424/back-from-the-summit-cephopenstack-integration/, November 2013.

[6] Ciscon, "Growth of Global Data Center Relevance and Traffic," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.html/.

[7] G. Coulouris, J. Dollimore, T. Kingberg, and G. Blair, “Distributed Systems: Concepts and Design,” 5th Edition, Pearson, 2012.

[8] Zhu Rongze, Haomai Wang, “Build an High-Performance and High-Durable Block Storage Service,” in openstack summit, Paris, France, 2014.

[9] Mosharaf Chowdhury, Srikanth Kandula, Ion Stoica, “Leveraging endpoint flexibility in data-intensive clusters,” Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM, Hong Kong, pp. 231–242, August 2013.

[10] Rajkumar Kettimuthu, Gayane Vardoyan, Gagan Agrawal, P. Sadayappan, Ian Foster, “An elegant sufficiency: load-aware differentiated scheduling of data transfers,” Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, USA, November 2015.

[11] Youngjae Kim, Scott Atchley, and Geoffroy R. Vallée, Oak Ridge National Laboratory, “LADS: Optimizing Data Transfers Using Layout-Aware Data Scheduling,” 13th USENIX Conference on File and Storage Technologies, pp. 67–80, 2015.

[12] Fio benchmark, http://linux.die.net/man/1/fio/.