Cluster Computing (2019) 22:373–384
https://doi.org/10.1007/s10586-018-2832-5

Async-LCAM: a lock contention aware messenger for Ceph distributed storage system

Bodon Jeong[1] · Awais Khan[1] · Sungyong Park[1]

Received: 19 March 2018 / Accepted: 17 July 2018 / Published online: 24 July 2018

Abstract
Ceph, an object-based distributed storage system, has a communication subsystem called *Async* messenger. In the *Async* messenger, a worker thread in the thread pool is assigned to each connection in a round-robin fashion and is allowed to process all the incoming or outgoing messages from the connection. Although this thread per connection strategy is easy to implement, it has an inherent problem such that when a connection is overloaded, it results in load imbalance problem among worker threads. In order to mitigate this problem, multiple worker threads can be assigned to a single connection to handle the traffic from the connection. However, this mapping structure induces another overhead related to lock contention since multiple threads contend to access the shared resources in the connection. In this paper, we propose lock contention aware messenger (*Async-LCAM*) , a messenger that assigns multiple worker threads per connection and is aware of lock contention generated from the threads. By keeping track of the lock contention of each connection every interval, the *Async-LCAM* dynamically adds or deletes assigned threads to/from the connection in order to balance the workloads among worker threads. The experimental results show that the *Async-LCAM* improves the throughput and latency of Ceph storage by up to 184 and 65%, respectively, compared to the original *Async* messenger.

Keywords Distributed storage system · Ceph · Thread placement · Load balancing · Lock contention · Epoll

1 Introduction

With the rapid proliferation of social network services and the increasing number of Internet of Things (IOT) devices, the amount of global digital information is increasing explosively every year. The market research agency IDC has predicted that the available data will increase up to 180 ZB by 2025 [1]. As the volume of data continues to grow exponentially, there has been a growing interest in the efficient management and storage of data. To accommodate such growing needs, distributed storage systems such as Ceph [2], Gluster [3], and Lustre [4] have emerged.

The distributed storage system spans over a large number of storage nodes. Such storage system creates volumes for each node and uses volumes as object storage server. Each object storage server performs high data networking through its communication layer. Gluster has a storage brick (storage server), and each storage brick performs data networking through the transport layer [3]. Lustre has an object storage server (OSS) and its I/O operations are sent over a network using a protocol called LNet [4]. Ceph has an object storage daemon (OSD) and a communication layer called messenger [2]. The distributed storage systems mentioned above highly count on the communication framework in order to guarantee the performance, scalability, and fault-tolerance.

Ceph uses *Async* messenger as a default communication framework to facilitate I/O traffic in and out of the cluster. The *Async* messenger is responsible for communication between clients and OSDs and within OSDs.

The *Async* messenger has a thread-pool structure that consists of a predefined number of worker threads [5]. When a request to create a connection is received from a

✉ Sungyong Park
parksy@sogang.ac.kr

Bodon Jeong
vodoni@sogang.ac.kr

Awais Khan
awais@sogang.ac.kr

[1] Department of Computer Science and Engineering, Sogang University, 35 Baekreom-ro, Mapo-gu, Seoul, Korea

client or OSD node, the *Async* messenger assigns one worker thread from the worker pool to the connection in a round-robin fashion. In other words, one worker thread is responsible for handling data traffic from one file descriptor (fd) associated with the connection. If the number of connections is greater than the size of the worker pool, one worker thread takes charge of more than one connection fd. Although this thread per connection strategy is easy to implement, it causes load imbalance problems among worker threads when each connection has a different workload that changes dynamically. That is, some threads can be idle, while other threads handle a large amount of data traffic. This uneven load distribution can cause performance degradation when there is high I/O traffic.

In this paper, we first perform an in-depth analysis of Ceph *Async* messenger and identify the limitations on the thread per connection structure that causes load imbalance among worker threads. Then, we propose lock contention aware messenger (*Async-LCAM*) that assigns multiple worker threads per single connection and is aware of lock contention in order to efficiently balance the workloads among worker threads.

We have implemented the *Async-LCAM* by extending the original *Async* messenger with Ceph community version 10.2.3. We used FIO benchmark [6] to evaluate the performance on a real testbed. The experimental results indicate that the *Async-LCAM* improves the throughput and latency of Ceph by up to 184 and 65% respectively, compared to the original *Async* messenger. To further show the impact of proposed optimization, we have also modified the source code of original Ceph such that the I/O data path is splitted into messenger, placement group (PG) processing, journaling and writing to filestore. Then, we analyze the performance bottlenecks by running the *Async-LCAM* and the original *Async* messenger over the modified data path. This paper has the following specific contributions :

- *Multiple worker threads per connection* The thread to fd mapping structure is changed so that all worker threads simultaneously monitor the data traffic from all the connections. This mapping structure enables fine-grain control of the data traffic and to maximize the redundant activities between connections, which reduces the load imbalance problems among worker threads.
- *Lock contention-aware thread placement* Assigning multiple worker threads per connection inherently induces lock contention overhead and diminishes the benefits of our new structure.The lock contention-aware thread placement algorithm proposed in this paper dynamically manages the mapping structure by keeping track of the lock contention of each connection every interval and dynamically adding or deleting threads to/from the connection.
- *Adjustment of the number of events a worker thread fetches* We also adjust the *maxevents* parameter in *epoll_wait* system call so that the number of fetched events in each worker thread is well balanced.

The rest of this paper is organized as follows. Chapter 2 outlines related work. In Chap. 3, we analyze the problems in current Ceph *Async* messenger and present the motivation of this paper. In Chap. 4, we discuss the structure of *Async-LCAM* and its detailed design. Chapter 5 provides performance evaluation with analysis. Chapter 6 concludes the paper.

2 Related work

The main contribution of *Async-LCAM* is the change of thread-to-fd mapping structure so that every thread monitors the traffic from all the connections and is aware of the lock contention in order to balance the workloads among worker threads. The rest of this section is devoted to a review of well-known approaches and systems designed for optimizing the communication subsystem in terms of load balancing and contention awareness.

There have been several studies focused on distributing load evenly among storage nodes in the clusters and cloud environments. Ceph [2] utilizes a reweight strategy to balance the load across OSDs. This strategy periodically and automatically balances data distribution by reweighting OSDs with high or low utilization. DLR [7] is a system that improves the performance of cloud storage services in the presence of hardware heterogeneity, and performance interference through a dynamic load redistribution technique. Sinbad [8] is a system that identifies network imbalance through periodic measurements and exploits the flexibility in endpoint placement to navigate around congested links. This improves end-to-end completion times of data-intesive jobs. SEAL [9] uses runtime information and data-driven models to approximate system load and adapts transfer schedules and concurrency so as to maximize performance while avoiding saturation. STEAL [9] uses user-supplied transfer types to further optimize schedules. STEAL treats batch and interactive transfers differently, allocating bandwidth unused by interactive transfers to batch transfers while being largely non-intrusive to interactive transfers. LADS [10] is a new bulk data movement framework for terabit networks among geographically distributed data centers. LADS implements layout-aware and congestion-aware scheduling strategies to minimize the effects of transient congestion within a subset of storage servers.

Shuffling [11] is a framework which migrates the threads of a multithreaded program across sockets to reduce the overhead caused by transferring locks between sockets. Shuffling reduces the time threads spend on acquiring locks and speeds up the execution of shared data accesses in the critical section. CA-scheduler [12] is a contention-aware scheduler that maps threads sharing the same lock-protected resources to the same processor. It presents a critical section-first scheduling strategy, which considers lock usage as a scheduling criterion to further reduce the thread waiting time due to lock contention.

The load imbalance problem in Ceph has also been addressed in studies by Han et al. [13] and Song et al. [14] through the genetic algorithm (GA) and the heuristic algorithm. Both studies maintain the original mapping structure (one thread per connection) of the *Async* messenger and periodically replace the worker thread assigned to one connection with another thread, while monitoring the data traffic handled by each worker thread. Although these two approaches also target at balancing the workloads among worker threads, the balancing is done on a per-connection basis. On the contrary, our approach provides a finer-grained control of the incoming traffic by assigning multiple threads per connection and using the lock contention-aware technique. Our study is based on the motivation to reduce the performance degradation via improving the short-comings of the mapping structure in the *Async* messenger and eliminating the load imbalance as well as possible contention across worker threads.

3 Background and motivation

In this chapter, we present the details of *epoll* mechanism and the overall architecture of the *Async* messenger in Ceph. We also discuss and analyze the problems in the *Async* messenger. For this, we conduct preliminary experiments with three Ceph clients, each of which generates random read/write traffic using the FIO benchmark [6]. The Ceph cluster equipped with four object storage servers (OSS) is used for the experiments, where each OSS has two OSDs.

3.1 Epoll mechanism

Event-driven approaches such as *select*, *poll*, and *epoll* [15, 16] have been widely used in high-performance networks to multiplex a large number of concurrent connections. Among them, *epoll* is a mechanism used in Linux system for a scalable I/O event notification mechanism, which is known to be more scalable than *select* or *poll* when monitoring a large number of connection fds.

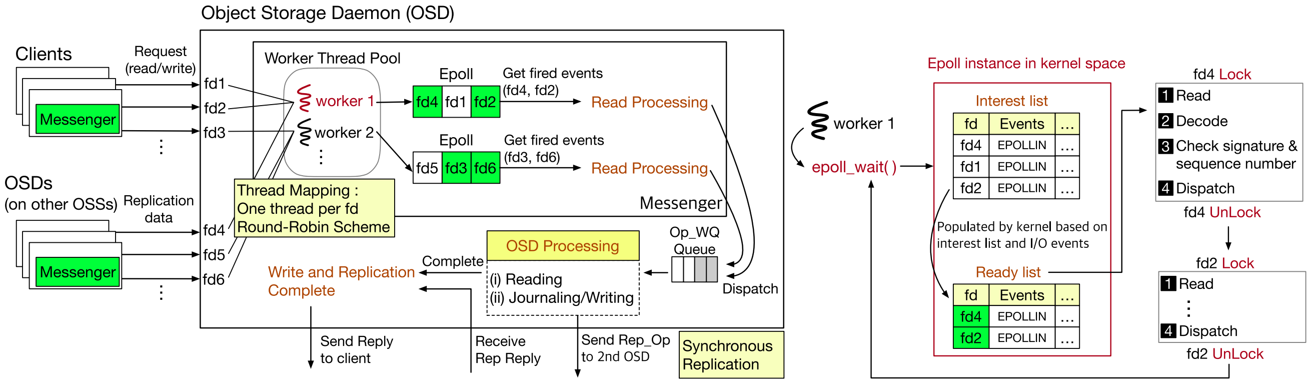
In *epoll* mechanism, three system calls are provided to set up and control an *epoll set*. The *epoll_create()* system call instructs the kernel to create an event data structure used to track events. The *epoll_ctl()* system call registers or deletes file descriptors that it is interested in. Finally, the *epoll_wait()* system call is used to receive the incoming events from the file descriptors. Once any events are detected, it returns the number of file descriptors ready for the requested I/O. As with BSD-like socket, the *epoll* mechanism does packet I/Os through the kernel TCP layer. The TCP socket is abstracted by virtual file system (VFS) and exposed to user-level as a socket file descriptor. Therefore, fd or connection fd mentioned in this paper refers to socket file descriptor.

3.2 Ceph Async messenger communication subsystem

Figure 1a depicts the overall communication flow of the *Async* messenger in Ceph storage system. All major components such as Ceph clients, OSDs, and Monitors use the *Async* messenger as a default socket-based communication carrier in Ceph.

As shown in Fig. 1a, a worker thread pool consists of a predefined number of threads that are created initially. The size of a worker thread pool is a tunable parameter and can be set as required. At first, a connection is required to communicate among components. When a request for creating a new connection is received, a worker thread from the worker thread pool is assigned to the connection in a round-robin order. Because the pool size is predetermined, if the number of connections is greater than the pool size, it is possible that single worker thread can be assigned to multiple connections. Still, in terms of connections, only one worker thread is assigned per connection. The worker thread assigned to a connection waits for any events from the connection using *epoll_wait()* system call in Linux. Finally, when the worker thread receives an event, it carries out a series of actions such as decoding the received data, checking signature and sequence number, and then inserts the processed event into an OSD queue called *Operation WQ*. After this, each worker thread repeats the whole process again.

If the received event is a write request, the worker thread initiates a replication request to secondary OSDs. At the same time, it also performs journaling and disk write on OSD. Then, if the worker thread receives an acknowledgment message indicating that replication operation is completed on the secondary OSDs, it sends an acknowledgment message back to the client. Whereas, in the case



(a) Overall Communication Flow **(b)** Processing Flow of a Worker Thread

Fig. 1 Communication process using *Async* messenger in Ceph

of a read request, it accesses the disk and transmits the required data directly to the client.

On the other hand, the complete processing flow of a worker thread is depicted in Fig. 1b. The worker 1 first creates an internal kernel data structure called *epoll instance* through *epoll_create()* system call. When a connection is established, the worker 1 registers the fd of the corresponding connection in its own interest list of the *epoll instance* via *epoll_ctl()* system call. For example, as shown in Fig. 1b, the worker 1 contains fd4, fd1, and fd2 in the interest list. The fd4 is at the top of the interest list because its request to create a connection comes first. If the event causes the fd to become ready, the kernel places the fd on the ready list of the *epoll instance*. Figure 1b shows that both fd4 and fd2 are ready to receive events. Therefore, the worker thread fetches the I/O events from those fds through *epoll_wait()* system call. The fetched events will be processed sequentially. In this example, the worker 1 reads data from fd4, decodes the received data, and confirms the data validity by checking signature and sequence number. Then, it dispatches the data to the OSD queue and repeats those activities for fd2 again. Current *Async* messenger uses a mutex lock to prevent the activities starting from read operation till dispatch operation from overlapping, which is in fact unnecessary because only one worker thread can process the traffic from one fd. However, the lock for each fd is necessary when the proposed *Async-LCAM* is employed in the communication subsystem of Ceph.

3.3 Load imbalance of worker threads in Async messenger

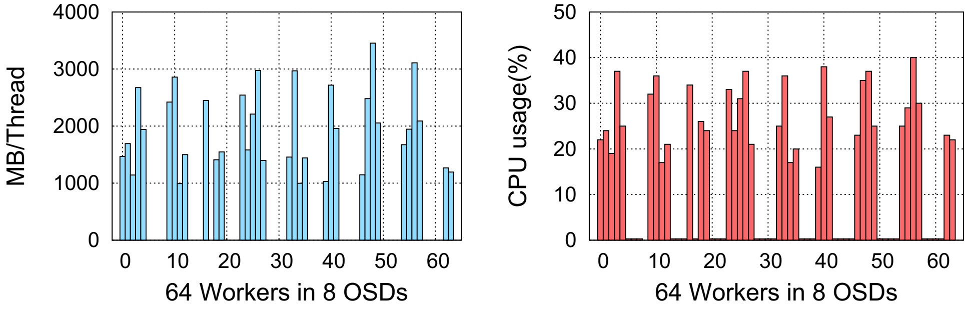
The *Async* messenger assigns a worker thread to one connection fd based on the round-robin approach. Because the round-robin approach assigns worker threads statically without reflecting the workload of each connection, some worker threads are possibly assigned to connection fds handling low-traffic messages such as OSD_PING, OSD_PG_INFO, and OSD_PG_NOTIFY, while other worker threads are assigned to high-traffic messages containing actual data such as OSD_OP and OSD_REPOP. In the worst case, some worker threads are idle and other threads handle connections with heavy traffic. This mapping structure and round-robin assignment of the *Async* messenger cause the bottleneck in a distributed environment, in particular as the number of connections is getting bigger.

In order to see how much imbalance each worker thread generates, we have measured the total size of messages handled and the average CPU usage of 64 worker threads. For this, random write operations with 4K block size were generated for 180 s. As shown in Fig. 2a, the number of messages handled by each worker thread is highly imbalanced as expected. For example, a few worker threads handle messages of more than 3000 MB for 180 s, while others handle messages of less than 1000 MB. Figure 2b also depicts the CPU usage with respect to the messages handled. The average CPU usages of some worker threads are over 30%, whereas those of others are far less than 20%.

3.4 Lock contention

A simple approach to solving the load imbalance problem discussed in Sect. 3.3 is to alter the mapping structure of worker thread to connection fd so that multiple worker threads are allowed to be assigned to a single connection fd. In such case, all worker threads can process messages in a more timely and load-balanced manner because they are always ready to receive the traffic from the connection.

Currently, in the original *Async* messenger, there exists no lock contention problem because only a single worker thread is assigned to one fd. In the case of the proposed



(a) Total Size of Messages **(b)** AverageCPU Usage

Fig. 2 Load imbalance problem in *Async* messenger

mapping structure, the chance of lock contention is highly likely when multiple worker threads try to handle the events for one fd concurrently. Because each worker thread has the same *epoll* interest list and the events are processed based on the order of this list, a certain notable lock contention occurs in specific fds. For example, Fig. 3 shows a case where the lock contention at a specific fd occurs in the proposed mapping structure. It is assumed that each worker thread has the *epoll* interest list consisting of fd1, fd2, and fd3 and the I/O traffic is generated continuously from fd1 to fd3. In this case, each worker thread is supposed to retrieve I/O events from those fds consecutively. As shown in Fig. 3, the worker 1 first locks fd1 and handles the event. Hence, worker 2 and worker 3 need to wait until the handling of the fd1 event by worker 1 is finished. In this example, lock contention occurs only for fd1, and not for fd2 and fd3.

In order to analyze this further, we have conducted another experiment and measured the times taken to obtain the locks in 30 connections. For the experiment, random write traffic was generated with 4K block size and the results were obtained for an interval of 60 s. Let us assume that $T_i(fd)$ is the lock time of the ith worker thread designated to a specific fd during the interval, where the lock time is the difference between the time when the worker thread arrives at the lock and the time when it obtains the lock. Thus, the total lock time of a specific fd is $\sum_{i=1}^{N} T_i(fd)$, where N is the number of worker threads currently assigned to the fd. As shown in Fig. 4, the total lock times of most fd connections are less than 5 s, while some fd connections take more than 30 s. This is because some worker threads take more time to acquire locks for the specific fds in which other worker threads have already acquired and are currently processing the events. This happened in fd1 case shown in Fig. 3. Therefore, the proposed mapping structure should be carefully designed so that multiple worker threads are dynamically assigned to a specific connection fd by considering the lock contention imposed on this connection fd.

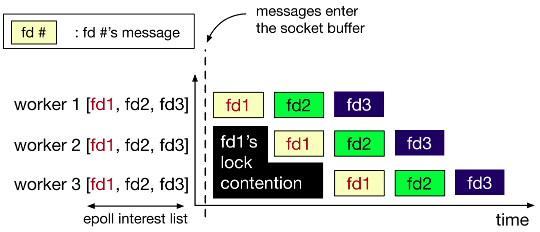


Fig. 3 Example of lock contention in the mapping structure (multiple threads per fd)

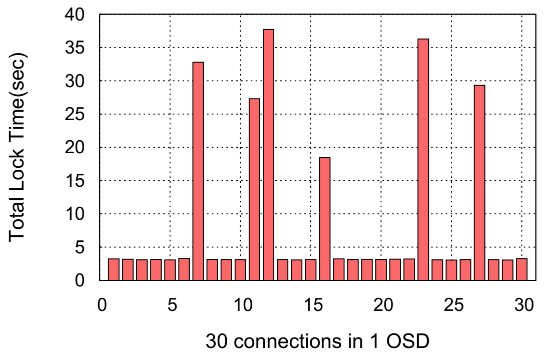


Fig. 4 Total lock time of each connection (multiple threads per fd)

3.5 Maxevents parameter

When a worker thread invokes the *epoll_wait()* system call, it passes a parameter called *maxevents*. This parameter indicates the maximum number of fds that can be retrieved simultaneously from the *epoll* ready list. The default value for *maxevents* in Ceph is 5000. In the original *Async* messenger, this default value does not cause any significant problems because each fd can be handled only by one worker thread and the number of events that can be fetched simultaneously from single fd is not so large. However, in the proposed mapping structure where each worker thread handles events for multiple fds, the possibility of having different number of events among worker threads increases.

Figure 5 shows the variation in the size of fd list obtained by one worker thread when random write traffic is

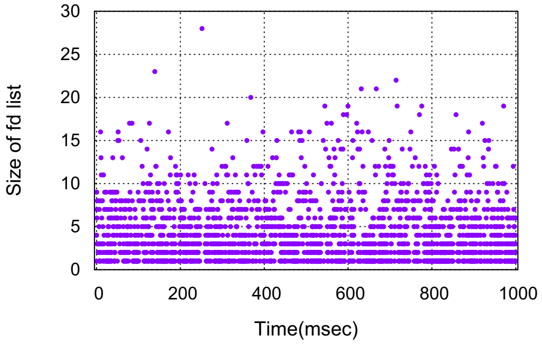


Fig. 5 Size of fd list fetched by the worker thread (multiple threads per fd)

generated with 4K block size for 1000 msec and the *maxevents* parameter is set to 5000. From the experiment, it is observed that some worker threads fetch three fds on average when calling *epoll_wait()*, whereas others get up to 28 fds. This means that one worker thread can fetch the events of almost all connection fds, which may create load imbalance problems among worker threads. Therefore, the proposed mapping structure needs to be designed in such a way that all worker threads can fetch equal-sized fd list by adjusting the *maxevents* parameter.

4 Design and implementation

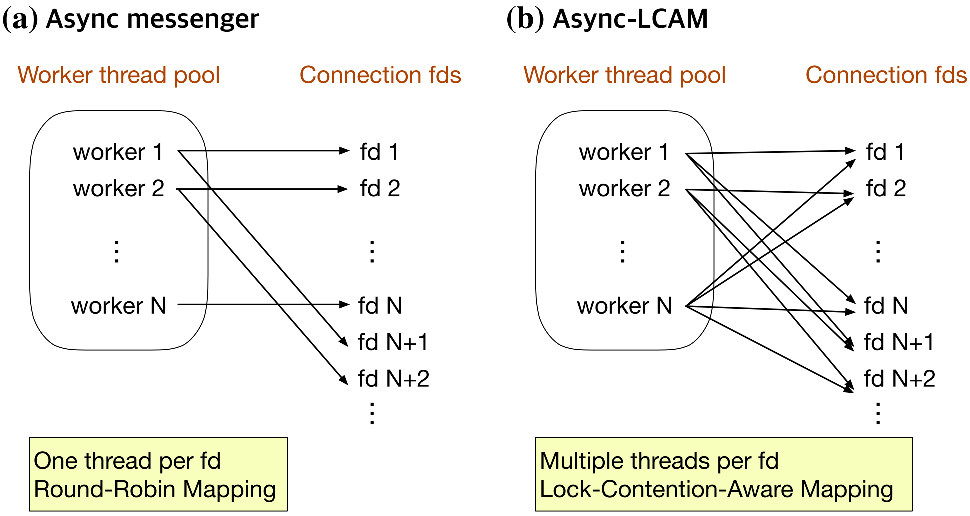
To address the load imbalance and lock contention problems, we propose an idea to change the thread to fd mapping structure of current *Async* messenger so that every worker thread monitors the traffic from all connections and is aware of the lock contention to efficiently balance the workloads among worker threads. In this chapter, we first explain the design and implementation issues of the proposed communication subsystem, *Async-LCAM*. Second, we present a lock contention aware thread placement algorithm to minimize the contention across multiple worker threads. Third, we analyze the impact of *maxevents* parameter in the *Async-LCAM*.

4.1 Mapping structure of Async-LCAM

In contrast to the original *Async* messenger, which assigns one worker thread per connection fd, the *Async-LCAM* assigns multiple worker threads to each connection fd. Figure 6a, b show the mapping structures of the *Async* messenger and the *Async-LCAM*, respectively. The mapping structure of the *Async-LCAM* has two benefits compared to that of the *Async* messenger. First, it automatically solves the load imbalance problem caused by the *Async* messenger because multiple worker threads compete to handle messages for all fds. This allows the *Async-LCAM* to efficiently handle messages in a more timely and load-balanced manner. Second, the mapping structure maximizes the redundant activities between connections.

The benefit of the *Async-LCAM* is further depicted in Figure 7a, b. Let us assume that the worker thread 1 is assigned to both fd1 and fd2 in the *Async* messenger. In this case, if a new event from fd2 occurs while the worker thread 1 is handling a message from fd1, the worker thread 1 has to handle the message from fd1 first before calling *epoll_wait()* to obtain and handle the message from fd2. This is because one worker thread is assigned only to the corresponding fd and it has to process the received events consecutively if multiple events occur at the same time. On the other hand, in the *Async-LCAM* shown in Fig. 7b, while the worker thread 1 is processing the message from fd1, the worker thread 2 is given an opportunity to handle the event from fd2. Thus, the performance can be improved as compared to the original *Async* messenger.

Fig. 6 Mapping structure of *Async* messenger and *Async-LCAM*



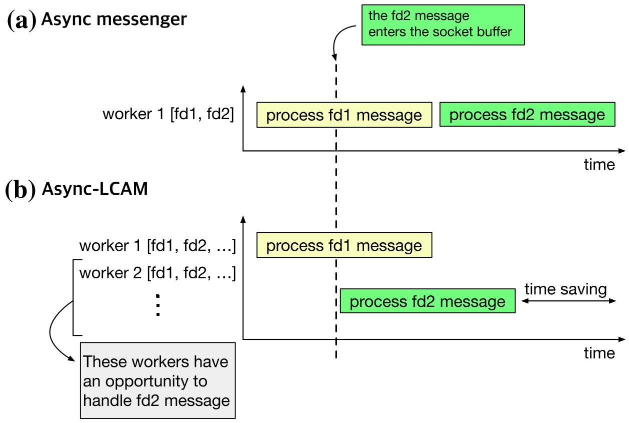


Fig. 7 Benefits of *Async-LCAM* compared to *Async* messenger

However, it should be noted that if multiple threads are assigned to a single connection fd, there is a possibility of lock contention. That is, the *Async* messenger does not need a lock for each fd because the message for one fd is handled only by one worker thread. Whereas, the *Async-LCAM* requires a lock for each fd because multiple worker threads try to handle different messages from the same fd. It is worthy to note that the fd connection is a message-critical structure handling the sequence from read operation to dispatch operation, the shared data structure can be corrupted if different messages of the same fd are handled simultaneously. Therefore, in order to mitigate the overhead caused by the lock contention, the *Async-LCAM* dynamically assigns multiple threads to the connection fd by considering lock contention.

4.2 Thread placement in Async-LCAM

As described earlier, the proposed mapping structure has the lock contention problem. To resolve this issue, we propose a lock contention-aware thread placement algorithm in this section. The proposed algorithm dynamically reduces the number of worker threads assigned to fds with large lock contention and adds additional worker threads to fds with less lock contention. This approach effectively controls worker threads while managing the mapping structure of the *Async-LCAM* that designates multiple worker threads to one fd. As a result, the lock time as well as total I/O time is decreased.

The proposed lock contention-aware thread placement algorithm is composed of three steps as shown in Algorithm 1. This algorithm is regularly executed by a special thread in the *Async-LCAM* in which the interval is set to 3 s.When the algorithm is ready to run, it first determines the total lock times of all fds during the interval (step 1). Then, if the total lock time exceeds a certain threshold, the algorithm performs a thread deletion process (step 2). Otherwise, the thread selection and addition processes are performed (step 3). The threshold value is determined empirically in our cluster environment and is currently set to 10% of the monitoring interval. In what folows, we explain the three steps in detail.

Step 1 *Monitor Threads* Each connection fd maintains a list of worker threads that are currently assigned to this connection. The total lock time is calculated by adding up the lock times of all worker threads in the list. The lock time of each worker thread is the accumulation of individual lock time during the interval, where each individual lock time is the difference between the time when the worker thread arrives at the fd lock and the time when it acquires the fd lock. If the total lock time is greater than the predefined threshold, the delete thread step (step 2) is executed. Whereas, if it is smaller, the select and add thread step (step 3) is performed.

Algorithm 1 : Lock Contention-Aware Thread Placement in Async-LCAM

```
$fd$ : the connection's fd
$T$ : the total lock time of $fd$
$N$ : the size of thread list currently placed on $fd$

procedure THREAD PLACEMENT
    Monitor Threads – calculate total lock time of
    $N$ threads.
    $T \leftarrow \sum_{i=1}^{N} T_i(fd)$
    if $T > threshold$ then
        Delete Thread – remove the $fd$ from the epoll
        interest list in the most recently placed thread.
    else
        Select Thread – select a thread that processes
        the least bytes.
        Add Thread – add the $fd$ to the epoll interest
        list of the selected thread.
    end if
end procedure
```

Step 2 *Delete Thread* This step is performed if the lock time of the corresponding fd is higher than the threshold value chosen earlier. Among the worker threads assigned to the connection, the most recently assigned worker thread is selected as a victim. Therefore, the connection fd with higher lock time than the threshold is removed from the *epoll* interest list of the victim worker thread so that it can no longer manage the messages from this fd. For example, Fig. 8a shows the process of removing the thread from the fd1 connection. In this figure, the recently assigned worker 3 is deleted from the thread list managed by fd1 and fd1 is deregistered from the *epoll* interest list of worker 3. After this step, the worker 3 is no longer responsible for handling the messages from fd1.

Step 3 *Select and Add Thread* This process is carried out if the lock time of the corresponding fd is found to be smaller than the threshold. In this step, an additional worker thread is assigned to the connection to maximize

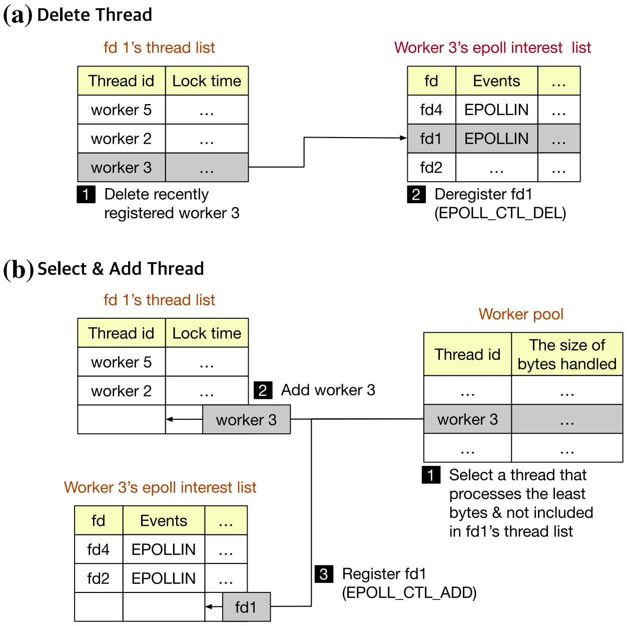


Fig. 8 Two Processes in the thread placement algorithm

the benefit from the proposed mapping structure. For this, we select a worker thread with the smallest amount of processed data as a candidate. Then the selected worker thread is assigned to this connection fd, which means that the connection fd is added to the *epoll* interest list of the selected thread so that it can handle the messages from this fd. Figure 8b shows the process of assigning an additional thread to the fd1 connection. Among the worker threads that have not been assigned to fd1, a thread with the smallest processed bytes is selected. In this example, it is assumed that the worker 3 is selected. Then, the worker 3 is added to the thread list managed by fd1, and fd1 is registered to the *epoll* interest list of worker 3. After this, the worker 3 can handle the messages from fd1.

4.3 Maxevents parameter

In Sect. 3.5, we highlighted that all worker threads in the *Async-LCAM* need to fetch equal-sized fd list so that the number of events is almost evenly distributed among worker threads. It should be noted that the maximum number of fds that one worker thread can retrieve after calling *epoll_wait()* is the total number of connections currently mapped to the worker pool in case the *maxevents* value is larger than the number of fds. Therefore, if the *maxevents* value is set to the number of current connections mapped to the worker pool divided by the pool size, the worker threads can equally handle incoming events.

Figure 9 shows an example of event handling processes in the worker threads when the events from fd1 to fd6 are generated constantly and each worker thread fetches events with different *maxevents* values such as 5000 and 2. It is also assumed that the thread pool size is 3. Figure 9a illustrates the process where the *maxevents* value is set to 5000, where one worker thread monopolizes the events from all fds. Whereas in Figure 9b where the *maxevents* value is set to the number of connections divided by the worker pool size, each worker thread can equally handle events from all fds. Although there is no difference in incoming workload among threads (each worker handles 6 events), the size of the fd list can be controlled by adjusting the *maxevents* value in the *epoll_wait()* system call. This allows the possibility of overlapping event processing activities among worker threads to be maximized, which leads to performance improvement.

5 Evaluation

In this chapter, we compare the performance of the *Async-LCAM* with two different *Async* messengers such as original *Async* messenger and *Async-TAM*. The *Async-TAM* refers to the *Async* messenger with load balancing capability based on the traffic-aware algorithm proposed in [14]. We also show the performance impacts of load balancing and lock contention-aware thread placement algorithm proposed in the *Async-LCAM*. For the evaluation, we compare input output operations per second (IOPS) and latency by using three different random workloads such as random read, random write, and random mix (i.e., mixed workload with 80% random read and 20% random write). Table 1 summarizes the description of each messenger version and its mapping structure.

5.1 Experimental setup

For the experiments, we configured Ceph cluster with four object storage servers (OSS) and each OSS has two SSDs acting as object storage daemons (OSD). We used three

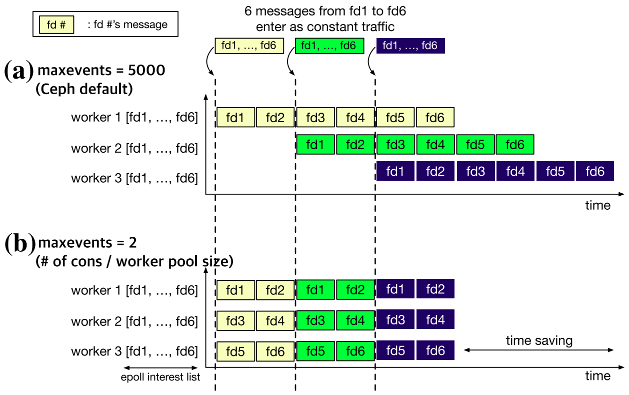


Fig. 9 Processing events in *Async-LCAM* according to *maxevents* value

Table 1 Types of messengers for comparison

Version	Description	Mapping structure
Async-Orig	Ceph-10.2.3 community *Async* messenger	One thread per fd
Async-TAM	A traffic aware algorithm in *Async* messenger [14]	One thread per fd
Async-LCAM	The proposed messenger in this paper	Multiple threads per fd

physical Ceph client nodes. The detailed specification of a server running OSD and Ceph client is summarized in Table 2. The replication factor and the size of worker thread pool were set to 2 and 8, respectively. In the thread placement algorithm, we set the monitoring interval to 3 s and the threshold value to 10% of this interval. We used FIO [6] benchmark with librbd/krbd support to vary block size and to generate different I/O traffic patterns.

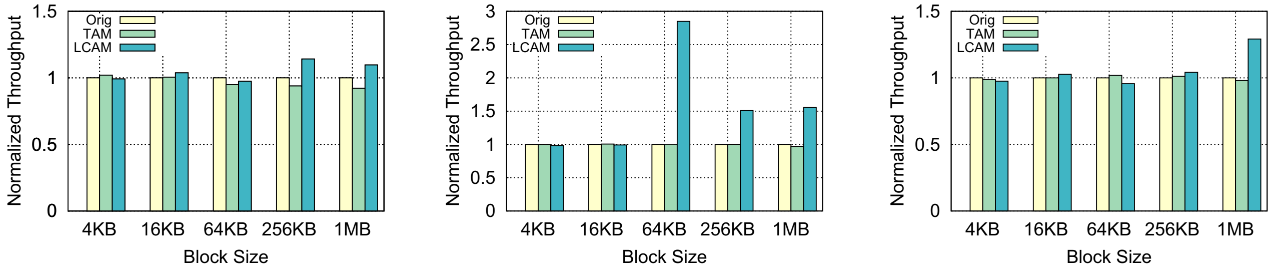
5.2 Performance evaluation

Figures 10 and 11 show the throughputs and latencies of three messengers listed in Table 1. We varied the block size from 4KB to 1MB and used three workload patterns such as random read, random write, and a mix of random read and random write (80% read and 20% write). We set the *iodepth* parameter to 128 and *numjobs* to 8 in FIO benchmark in order to generate high I/O traffic in the cluster.

As shown in Figs. 10a and 11a, the *Async-LCAM* outperforms the *Async-Orig* in random write workload when the block size is large. The *Async-LCAM* improves the throughput by 4 and 10% for block sizes of 256 KB and 1 MB as compared to the *Async-Orig*. The same trend is observed for latency with an improvement of 13 and 9% with block sizes of 256 KB and 1 MB, respectively. However, the *Async-TAM* shows no improvement in throughput and latency as reported in [14]. In random read workload, the *Async-LCAM* also outperforms the *Async-Orig* for relatively large block sizes as shown in Figs. 10b and 11b. For example, the *Async-LCAM* improves the throughput by a factor of 184, 50 and 55% for block sizes of 64 KB, 256 KB and 1 MB when compared against the *Async-Orig*. Furthermore, the *Async-LCAM* also reduces the latency up to 65, 34 and 36% for 64 KB, 256 KB and 1 MB block sizes with respect to the *Async-Orig*. These throughput and latency gains are due to the revised mapping structure and contention-aware thread placement in the *Async-LCAM*. Figures 10c and 11c compare the throughput and latency in random mix workload. Compared to the *Async-Orig*, the *Async-LCAM* improves the

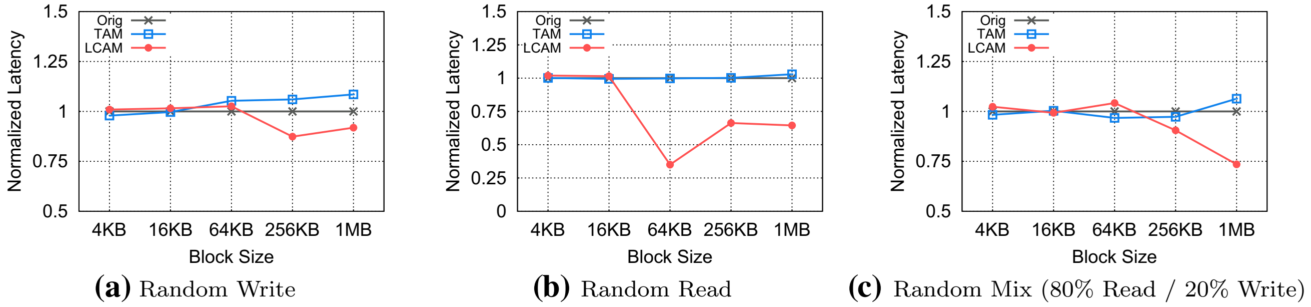
Table 2 Experimental setup

Client node (x3)	
Processor	Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GH (4 cores)
Memory	32GB
OS	CentOS 7.3.1611 (Kernel version 3.10.0-514)
OSS node (x4)	
Processor	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz (10 cores)
Memory	32 GB
Network	10 Gbps
OS	CentOS 7.3.1611 (Kernel version 3.10.0-514)
Disk	Samsung SSD 850 PRO 256 GB * 2/OSD node



(a) Random Write **(b)** Random Read **(c)** Random Mix (80% Read / 20% Write)

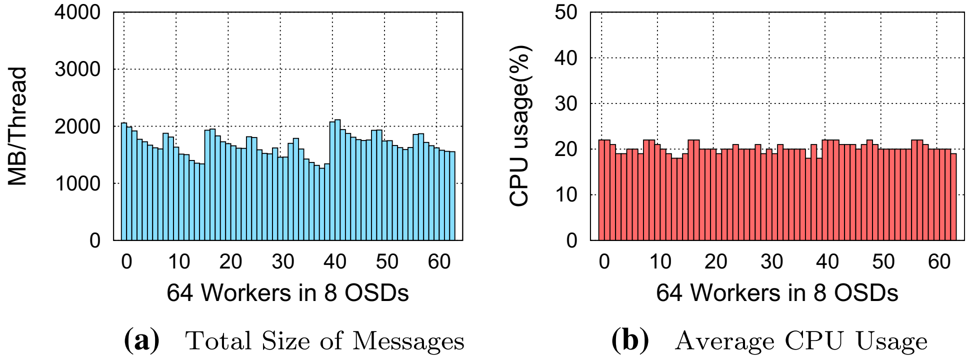
Fig. 10 Comparison of normalized throughput in random workloads



(a) Random Write (b) Random Read (c) Random Mix (80% Read / 20% Write)

Fig. 11 Comparison of normalized latency in random workloads

Fig. 12 Load balance of worker threads in *Async-LCAM*



(a) Total Size of Messages (b) Average CPU Usage

throughput by 5 and 29%, and the latency by a factor of 8 and 10% for block sizes of 256 KB and 1 MB.

Overall, the *Async-LCAM* shows notable performance improvement for large-sized blocks. This performance improvement is mainly attributed to the proposed mapping structure of assigning multiple worker threads to single connection file descriptor by considering lock contention overhead. The performance improvement for random write operation in the *Async-LCAM* is less than that of the read operation. This is because the Ceph client only requires the messenger layer to read the data from storage server. Whereas, in the case of write operation, the messenger receives the data from the client and then continues to the next layers such as PG processing, journaling and disk writing. These backend operations are proved to diminish most of the benefits of proposed idea as will be discussed in Sect. 5.4. It is also noteworthy that the performance improvement for small-sized blocks is almost negligible. This can be explained by the fact that the lock times for processing small-sized blocks are relatively smaller compared to those of large-sized blocks and thus the benefit of using lock contention-aware thread placement is minimized. Futhermore, the backend operations for write requests reduce the overall performance improvement. It is also observed that there is no vital performance improvement in the *Async-TAM*, which indicates that the *Async-LCAM* is more effective in distributing the workloads evenly across worker threads than *Async-TAM*.

5.3 Benefit analysis in Async-LCAM

In order to examine the efficacy of load balancing among worker threads in the *Async-LCAM*, we have again measured the total size of messages handled per thread and its corresponding average CPU usage with the same configurations as reported in Fig. 2. As shown in Fig. 12a, b, all worker threads in the *async-LCAM* process the similar number of messages within the range of 1500 and 2100 MB and the average CPU usage of each worker thread is approximately between 18 and 22%. This indicates that the workload distribution among threads is fairly balanced as compared to that of the *Async-Orig* shown in Fig. 2.

Another benefit of using *Async-LCAM* is its lock awareness. To show how well the proposed thread placement algorithm manages the lock contention occurred in the *Async-LCAM* mapping structure, we have conducted a similar experiment using the same configurations as we reported in Fig. 4. As shown in Fig. 13, the maximum lock time out of 30 connections by using the proposed lock contention-aware thread placement algorithm is only 5 s. This is a significant improvement compared to the result of *Async-Orig* (maximum 35 s) shown in Fig. 4. This improvement is mainly attributed to the lock contention-aware thread placement algorithm proposed in this paper.

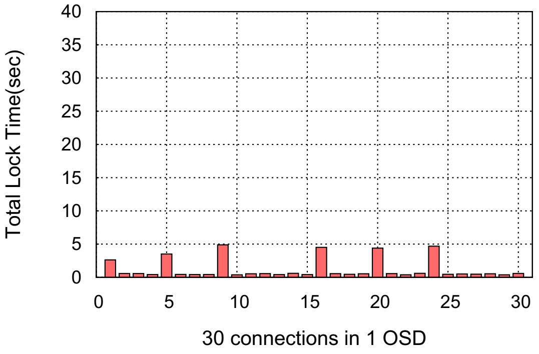


Fig. 13 Lock contention in *Async-LCAM*

5.4 Bottleneck analysis in PG processing

Unlike the read operation, the write operation in Ceph includes communications among multiple components, i.e., messenger layer and internal PG processing in OSD, journaling, and writing to filestore. Therefore, even if the performance of write operation is improved at the messenger layer, the entire storage performance may not be guaranteed due to the inclusion of several components.

To identify possible bottlenecks and where the performance degradation takes place, we modified the data path in Ceph. Figure 14 shows the two modified data paths: *Msgr Test* and *Msgr+PG Test*. The *Msgr Test* includes only the data path that returns its control as soon as each worker thread dispatches events to the OSD. The *Msgr+PG Test* includes the data path that continues its control until the PG processing in OSD layer is finished and skips journaling and wrting to filestore.

Figure 15 compares the IOPS value of *async-LCAM* with that of the *Async-Orig* in three different settings: *Msgr Test*, *Msgr+Pg Test*, and *Full Test*. The *Full Test* is a test that passes through all Ceph steps without any modifications. For the comparison, the random write workload with 4 KB block size is generated. As shown in Fig. 15, the *Async-LCAM* improves the throughput by 27% with respect to the *Async-Orig* in *Msgr Test*. However, no further throughput improvement is observed in *Msgr+Pg Test* and *Full Test*. This result indicates that the performance is degraded highly when the incoming messages pass through the PG processing step. Since the performance improvement in the messenger is canceled out from the PG processing overhead in OSD, the *Full Test* shows no further improvements. This explains the results discussed in Sect. 5.2 that show no performance improvements especially in small sized blocks.

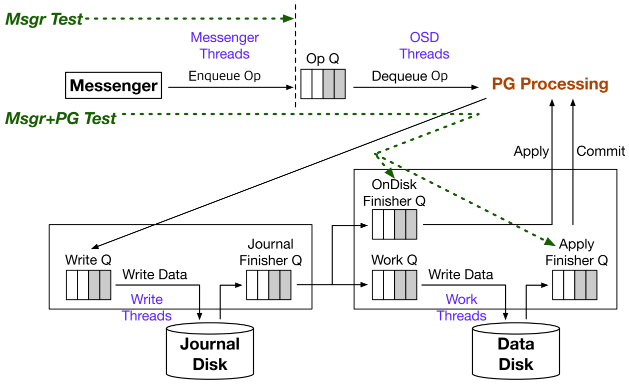


Fig. 14 Modified data path in Ceph

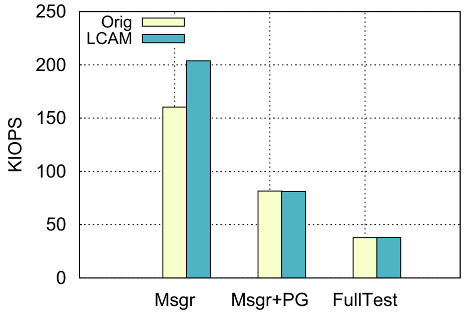


Fig. 15 Comparison of throughput in modified data path

6 Conclusion

Ceph uses a socket-based communication system called *Async* messenger to expedite client and internal cluster traffic, i.e., heartbeat, peering, and replication. The *Async* messenger assigns a worker thread into a single connection file descriptor. This one-to-one mapping structure creates asymmetric load distribution across worker threads. Such asymmetric and uneven load distribution results in performance degradation when there is high I/O traffic in the cluster. To address this uneven load distribution problem, we propose an idea to assign multiple threads to the per-connection file descriptor. However, such strategy of assigning multiple threads to a single connection file descriptor can cause lock contention resulting in increased latency for some I/O transactions.

In this paper, we present lock contention aware messenger (*Async-LCAM*), a Ceph messenger that evenly balances the load among worker threads. The *Async-LCAM* maximizes redundant activity between connections. It also dynamically determines the mapping structure between threads and connection fds, in a contention-aware manner via lock contention-aware thread placement algorithm. We have implemented the *Async-LCAM* in Ceph and compared its performance with those of baseline Ceph *Async* messenger and *Async-TAM* that implements a traffic-aware algorithm proposed in [14]. The evaluation resuts showed that the *Async-LCAM* improves the throughput and latency of Ceph storage by up to 184 and 65%, respectively, compared to the original *Async* messenger.

Acknowledgements This research was supported by the MSIT (Ministry of Science, ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2018-2016-0-00465) supervised by the IITP (Institute for Information & communications Technology Promotion).

References

1. IDC. https://www.ibm.com/blogs/internet-of-things/ai-future-iot/
2. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: 7th Symposium on Operating Systems Design and Implementation, USENIX Association, pp. 307–320 (2006)
3. Gluster. http://www.gluster.org/
4. Lustre. http://lustre.org/
5. Ling, Y., Mullen, T., Lin, X.: Analysis of optimal thread pool size. In: ACM SIGOPS Operating Systems Review, vol. 34, no. 2, April 2000, pp. 42–55 (2000)
6. Fio Benchmark. http://freecode.com/projects/fio
7. Noel, R.R., Lama, P.: Taming performance hotspots in cloud storage with dynamic load redistribution. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD). IEEE, pp. 42–49 (2017)
8. Chowdhury, M., Kandula, S., Stoica, I.: Leveraging endpoint flexibility in data-intensive clusters. In: ACM SIGCOMM Computer Communication Review. ACM, pp. 231–242 (2013)
9. Kettimuthu, R., Vardoyan, G., Agrawal, G., Sadayappan, P., Foster, I.: An elegant sufficiency: load-aware differentiated scheduling of data transfers. In: 2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, pp. 1–12 (2015)
10. Kim, Y., Atchley, S., Vallee, G.R., Lee, S., Shipman, G.M.: Optimizing end-to-end big data transfers over terabits network infrastructure. In: IEEE Transactions on Parallel and Distributed Systems, pp. 188–201 (2017)
11. Kumar, K., Rajiv, P., Laxmi, G., Bhuyan, N.: Shuffling: a framework for lock contention aware thread scheduling for multicore multiprocessor systems. In: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT). IEEE, pp. 289–300 (2014)
12. Xian, F., Srisaan, W., Jiang, H.: Contention-aware scheduler: unlocking execution parallelism in multithreaded java programs. In: ACM Sigplan Notices. ACM, pp. 163–180 (2008)
13. Han, Y., Lee, K., Park, S.: A Dynamic Message-aware Communication Scheduler for Ceph Storage System. In: IEEE International Workshops on Foundations and Applications of Self* Systems. IEEE, pp. 60–65 (2016)
14. Song, U., Jeong, B., Park, S., Lee, K.: Performance optimization of communication subsystem in scale-out distributed storage. In: 2017 IEEE 2nd International Workshops on Foundations and Applications of Self*Systems (FAS* W). IEEE, pp. 263–268 (2017)
15. Provos, N., Lever, C.: Scalable network I/O in linux. In: USENIX Annual Technical Conference, FREENIX Track (2000)
16. Chandra, A., Mosberger, D.: Scalability of linux event-dispatch mechanisms. In: USENIX Annual Technical Conference, Boston(2001)

Bodon Jeong is an employee of Samsung Electronics, Hwaseong, Kyongki-do, Korea. He received his B.S. and M.S. degrees in Computer Science and Engineering from Sogang University, Seoul, Korea. Now he works for Samsung Electronics as a software engineer.

Awais Khan is an integrated program student in Sogang University, Seoul, South Korea. He received his B.S. degree in Bioinformatics from Mohammad Ali Jinnah University, Islamabad, Pakistan. He worked for one of leading software companies as a software engineer from 2012 to 2015. Currently, he is a member in Laboratory for Advanced System Software at Sogang University Computer Science and Engineering department. His research interests include cloud computing, cluster-scale deduplication, parallel and distributed file systems.

Sungyong Park is a professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Korea. He received his B.S. degree in computer science from Sogang University, and both the M.S. and Ph.D. degrees in computer science from Syracuse University. From 1987 to 1992, he worked for LG Electronics, Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.