Dynamic Load Balancing of Dispatch Scheduling for Solid State Disks

Myung Hyun Jo, *Student Member, IEEE,* and Won Woo Ro, *Senior Member, IEEE Computer Society*

Abstract—Providing low-latency and high-throughput is an important design feature of an I/O scheduler. Especially, when multiple applications share and compete for a storage resource, the operating system is required to schedule the IO requests for the maximum throughput. Recently, the flash-based storage, solid-state drive (SSD) has been popularly used in various computing systems and traditional scheduling algorithms have been researched and tuned for the emerging flash-based storages. However, the SSDs suffer from the contention problem caused by multiple I/O requests and experience significant performance degradation. This is mainly due to the concurrently accesses to a finite set of flash memory chips. In this paper we propose *Dynamic Load Balanced Queuing* (DLBQ) that reorders the I/O requests and evenly distributes the accesses on flash memory chips to avoid contention. For that purpose, we have introduced a virtual time method which chases the run-time status of the SSD. We have evaluated the throughput and latency of DLBQ versus the four I/O schedulers with micro benchmarks and server benchmarks. The experimental results show that the throughput of DLBQ is improved by 11 % on a 128 GB SSD and 15 % on a 256 GB SSD while ensuring a bounded latency.

Index Terms—Scheduling, storage management, performance

1 Introduction

HIGH-speed storage such as flash-based solid-state drives (SSDs) have emerged and provided enhanced performance especially for data-centric systems. In fact, SSDs can provide high throughput by concurrently accessing multiple flash memory chips where data are striped over as in RAID-0 architecture [1], [2], [3]. This parallel accesses are originally expected to bring huge advantages for data-intensive applications such as web server, file server, and database [4]. However, the amount of data those applications normally access cannot fully exploit the parallelism of multiple flash chips; the data sizes are mostly as small as 8 or 16 KB [5], [6]. On the perspectives of flash-based storage, the small size data can normally cause frequent accesses to a set of specific memory chips under a static striping fashion (RAID-0) and consequently results in contention; the *contention* means that I/O requests are handled by only specific flash memory chips during a short time interval [5], [6], [7], [8]. The slack delay caused by the contention substantially decreases the storage utilization [9], [10], and hence significantly degrades the performance of the applications.

Traditionally, I/O scheduling algorithms for mechanical disks have been studied to guarantee the fairness and predefined latency [11], [12], [13], [14], [15], [16], [17]. To ensure fairness, the algorithms exploited time-based strategies such as virtual time [11] or budget [15], in which resource is proportionally allocated to the competing applications based on the predefined weights or service rates. In addition, a feedback control method keeps a desired latency by limiting the number of the I/O requests that can be simultaneously executed in a constant time interval [12], [14], [16]. However, the contemporary storages with a plurality of flash chips have experienced significant performance variation by the internal operations such as static striping, wear leveling, or garbage collection. Recent I/O schedulers have targeted to achieve high throughput of storage rather than ensuring fairness or bounded latency [5], [8], [18], [19], [20]. To minimize the response delay of read requests caused by the long write operations of flash memory, the algorithms separately handle read and write I/O requests [5], [18], [19], [21], [22]. Moreover, some scheduling algorithms have tried to avoid the conflict of the parallel resources within the storage such as flash memory chips [8], [20] or channels [5].

Although the recent I/O schedulers have overcome the constraints of flash-based storage such as the chip collision and the latency difference between read and write requests, the existing methods are insufficient to achieve both high throughput and low latency for the following three reasons. Firstly, the previous chip collision-avoidance dispatches a set of requests that do not access the same flash memory chips [5], [8]. This collision-avoidance is efficient if the request size is small so that each request accesses only a dedicated flash memory chip. Unfortunately, the requests of server workloads mostly access two or more flash memory chips with a random pattern. It implies that selectively gathering the non-competing requests is significantly difficult. Furthermore, this predicted collision may not coincide with the actual collision if an I/O scheduler does not recognize the completion of the dispatched requests. Unlike the conventional hard disk, the response time of requests handled on the flash-based storage can be longer than expected due to the internal operations such as garbage collection. Sec-

- *The authors are with the School of Electrical and Electronic Engineering, Yonsei University, Sinchon-dong, Seodaemun-gu, Seoul, Korea. E-mail: {myunghyun.jo, ws.jeong, wro}@yonsei.ac.kr*
- *M.H.Jo is also with the Software Development Team, Memory Division, Samsung Electronics, Hwasung, Gyeonggi-Do 445-701, Korea. E-mail: {myunghyun.jo}@samsung.com*

This paper is an extension of our previous study, "Contention-Free Fair Queuing for High-Speed Storage with RAID-0 Architecture," which appeared in the 2015 IEEE 17th International Conference on High Performance Computing and Communications.

ondly, the batch techniques of minimizing the interference between read and write requests within the storage are not explicitly optimized [5], [22]. The inter-mixing extent of read/write requests is considerably different depending on the sizes, patterns, and read/write ratios of workloads. To accurately ensure a constant interval in which read and write requests are not mixed, a batch method should be globally aware the chip access distribution in runtime. In addition, the scheduling ratio between read and write queues must be balanced for diverse workload sets or system environments. Finally, the previous algorithms do not attempt to bound the long-tail latency occurred by the request reordering, which is carried out for high performance.

To overcome these problems, we propose a new I/O scheduling algorithm, called *Dynamic Load Balanced Queuing (DLBQ)*, that reorders the I/O requests to maximize throughput of the flash-based storage while limiting the worst-case latency. Because the contention problems of read/write interference and chip collision incur the significant slack time of flash memory chips, DLBQ is designed to achieve both the contention-avoidance and high throughput. To solve the contention problem, DLBQ first defines the cost model that represents the load of the flash memory chips in a virtual time fashion. When an I/O request is dispatched, the virtual finish times of accessing flash memory chips are accumulated by the request length. If the dispatched request is completed, the virtual start times of associated flash chips are accumulated. By the difference between two virtual times, DLBQ can dynamically verify the load assigned to each flash chip.

Using the cost model, DLBQ applies two algorithms: the dynamic queue selection (DQS) and the balanced chip utilization (BCU). To prevent the mixing of the read/write requests in the storage, DQS chooses one of two queues dedicated to request types, i.e., read and write. Further, DQS evenly balances the activation time for dispatching between two queues, especially for guaranteeing the fairness of read requests. After deciding the target queue by DQS, BCU finds an I/O request that can improve the storage utilization among the backlogged requests in the selected queue. To estimate the utilization for each request, BCU exploits the ratio of the total load assigned to flash memory chips until the expected finish time of the request. In addition, BCU compensates the utilization for the waiting cost of the request which is pushed back from a dispatching order. Therefore, DLBQ can dispatch an I/O request with high utilization and bounded worst-case latency.

The rest of the paper is organized as follows. In Section 2, we present works related to our research. In addition, we describe the architecture and characteristics of flash-based storage. On the basis of the properties of flash-based storage, we present a new I/O scheduler to meet our goals in Section 3. In Section 4, we compare the throughput and latency between various I/O schedulers with a diverse set of workloads. Finally, Section 5 concludes the paper.

2 Background

This section first illustrates the structure and internal operations of flash-based storage. In addition, we present our research motivations.

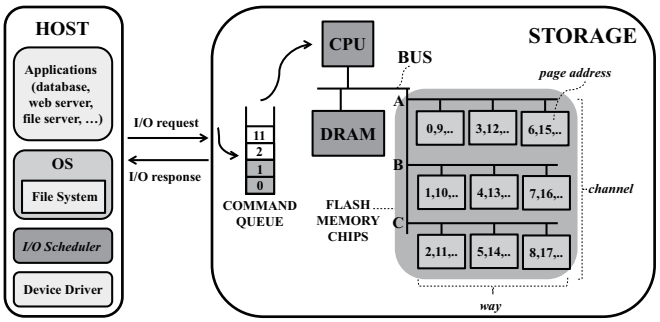


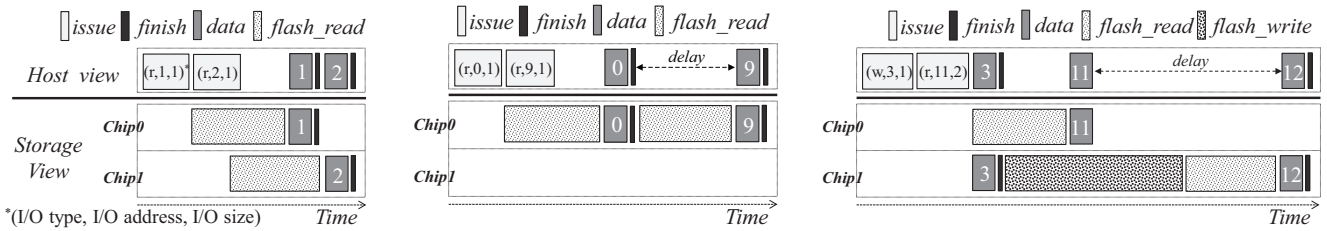
Fig. 1. Overview of flash-based storage

2.1 Flash-Based Storage

To represent the internal constraints of flash-based storage, we show the general architecture of flash-based storage in Fig. 1. The basic structure resembles typical computer systems, except the command queue and the multiple flash memory chips. CPU executes the embedded firmware with internal operations such as static striping, address translation, and wear leveling. DRAM is used for transmitting data from/to the host. A plurality of flash memory chips are statically partitioned and connected to multiple channels and ways; it can simultaneously load/store the request data from/to the flash memories. The command queue is equipped for enabling the asynchronous execution of multiple I/O requests.

In practical, the number and size of the components, i.e., CPU, DRAM, the command queue, and flash memory chips, are configured considering cost, power, and performance. The number of flash memory chips is determined by storage capacity. Moreover, the size of a command queue is limited to 32 by the storage specification such as SATA [23]. Accordingly, I/O requests from server applications are mostly waiting in the queue of I/O scheduler located on operating system. Because the server workloads are generally produced from more than 100 threads [24], the number of the requests passed at the same time far exceeds the size of command queue.

Here we present the internal operations related to the processing of I/O requests in the flash-based storage. First, I/O requests from the host are pushed into the command queue of storage. As shown in Fig. 1, four I/O requests with the address of 0, 1, 2, and 11 can be enqueued continually if the command queue is not full. Then, an embedded firmware sequentially pops a single request from the command queue. To handle the requests in parallel, the firmware performs an modular operation of request address according to the number of flash memory chips; in this paper, the address and length of I/O requests are configured as the page size of 4 KB because the processing (READ and PROGRAM) unit of flash memory is generally 4 KB. If the request size is larger than the striping size of 4 KB, the request is divided into the striping size. Then, the partitioned requests are distributed to the associated flash chips. In Fig. 1, the four requests are assigned to the first way of channel *A*, channel *B*, and channel *C* in the three-channel/three-way storage. The last two requests with the address of 2 and 11 are assigned to the same flash chip.



(a) Sequential pattern (b) Collision pattern of single page (c) Collision pattern of multiple pages

Fig. 2. Lifecycles of various I/O patterns in the flash-based storage

2.2 Motivation

Fig. 2(a) shows how parallel access in modern flash-based SSDs can improve the throughput. For example, two read requests with the logical page address of 1 and 2 are allocated to different flash memory chips and handled parallely. Assuming that the read operations of flash memory *Chip 0* and *Chip 1* are executed in parallel, the storage throughput is roughly linear to the number of flash memory chips.

However, the real throughput can be considerably varied according to the I/O patterns. As mentioned before, the flash-based storage statically distributes multiple I/O requests to flash memory chips according to the request address. Therefore, simultaneous accesses to the same flash chip can happen. As shown in Fig. 2(b), two read requests with the address of 0 and 9 are assigned to the same flash chip when the storage is constructed to nine chips. Then, the two read operations of flash memory *Chip 0* should be executed sequentially. Accordingly, the completion of the trailing read request is delayed.

Meanwhile, if the workloads have a random address pattern, the storage throughput does not decrease largely even if the chip collision of single page is occurred. Because the storage has sufficient multiple chips, e.g., 16 or 32, the other non-conflicted chips can be preferentially served. Then, the requests with single-page size can be completed immediately by finishing an associated flash operation. Accordingly, when the command queue is full, a new request can be entered into the storage if only one of multiple flash operations is completed. However, if an I/O request causes accesses to multiple pages, the frequency of chip collision would increase [6]. When the states of several flash memory chips are busy, there is a high possibility that the requests to multiple pages would access the busy chips. It will cause request conflicts and result in the long response time because the request cannot be completed until all pages are serviced.

Furthermore, the throughput can be also degraded due to the latency discrepancy between the read and write requests. For example, let us assume that a write request with the address of 3 and the size of 4 KB precedes a read request with the address of 11 and the size of 8 KB, as shown in Fig. 2(c). If the storage employs an internal write buffer, the data is first stored in the write buffer and moved to each corresponding flash chips. For the tailing read request, the first 4 KB data of the address 11 can be instantly read and sent to the host because the flash memory *Chip 0* is not blocked. However, the second 4 KB data of the address 12 is blocked until the previous write operation on *Chip 1* is finished. This scenario shows that the trailing read request can be significantly delayed due to the long write operation; in fact, the read delay is 50 μs whereas the write delay can be as large as 1 ms [25]. Moreover, in the storage specification of SATA [23], the storage cannot receive additional I/O requests during data transmission even if the command queue is not full. With this observations, we consider separation of read and write request handling for better scheduling.

These contention problems substantially increases the slack time of flash memories when a single request does not fully utilize all flash memory chips. Therefore, we target that the proposed I/O scheduler solves the contention problems when the flash-based storage is used for the server workloads that are composed of mixed, medium size, and many threads.

2.3 Related Work

I/O scheduling algorithms have been long studied to provide guaranteed quality-of-service (QoS) of storage with internal parallelism. Because multiple competing applications share limited resources, a scheduler should fairly divide the total bandwidth of storage into N applications with different weights. To ensure the proportional bandwidth allocation, SFQ(D) [11], Argon [13], PARDA [14], BFQ [15], and Maestro [16] have exploited the time-based strategies, which evenly allocate a virtual time or budget to each application.

In fact, an accurate time model is needed to achieve a fair distribution of bandwidth. However, the realistic time model will enforce computation intensive and repeated recalculation procedures every time when a request is made or selected for transmission [26]. To lessen this computation overhead, the concept of virtual time is introduced. The virtual time can allocate the bandwidth for each application without recalculating the actual finish time for previously queued requests [27]. At the virtual time model, a request easily can determine the virtual finish time by incrementing the request length to the virtual start time, when requests arrive at the queue dedicated to each application. Accordingly, a scheduler dispatches preferentially a request with the earliest virtual finish time among the application queues, and the bandwidth of each application is evenly distributed according to the request size.

In addition, Avatar [12], PARDA [14], and Cake [17] exploit adaptive feedback control to ensure the user-defined latency. By throttling the number of *outstanding* I/O requests, the schedulers adjust the number of concurrently

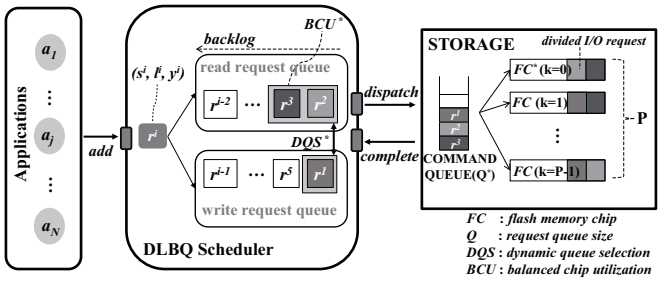


Fig. 3. The system model of DLBQ

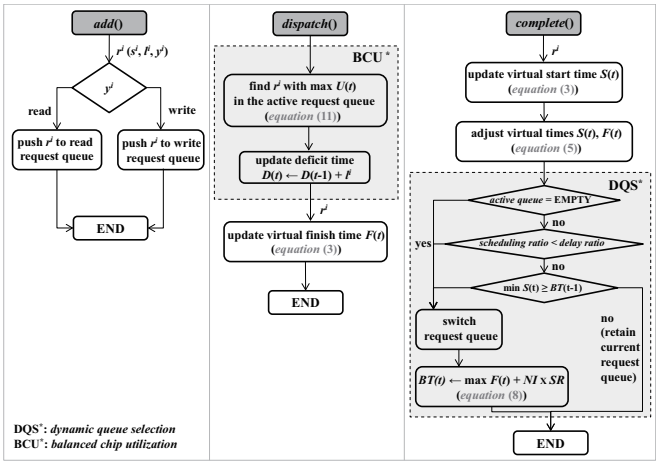


Fig. 4. The execution flow of DLBQ

activated resources within the storage; any requests dispatched from the I/O scheduler are *outstanding* if their completions have not received a response from the storage device. Guaranteeing the fairness and bounded latency is essential for ensuring the QoS for the shared storage. However, the traditional scheduling methods are not easy to improve the throughput of storage especially for flash-based storage, because they typically treat the storage system as a black box; PARDA [14] and BFQ [15] optimize only the overheads of mechanical disk drives, e.g., the seek time of a HDD.

As emerging the flash-based solid-state-drives (SSD), an I/O scheduling algorithm has been increased the opportunity to improve the storage performance. First, the algorithms have proposed a method of avoiding the read/write interference occurred by the latency discrepancy of flash memory [5], [18], [19], [21], [22]. With the separated queue per a request type, ParDispatcher [22] and PIQ [5] dispatch a set of requests with the same type in a batch fashion, whereas FIOS [21] and BCQ [8] preferentially dispatch a set of read requests. ParDispatcher uses a batch size predefined by a micro-benchmark [22] and PIQ distinguishes a batch of irrelevant requests according to a request type [5]. Though the previous algorithms solve a part of the read/write interference, these batch techniques are not enough to avoid the interference adaptively for diverse workloads or system environments. Second, BCQ [8], PIQ [5], and CFFQ [20] are trying to prevent the performance degradation caused by the contention of shared resources, i.e., channels or flash memory chips. BCQ splits I/O requests to an associated channel queue [8]. PIQ leverages the conflict vector that records the chip access of requests to collect multiple irrelevant requests into a single batch [5]. The our previous research, CFFQ, exploits the device-oriented virtual time that represents the load assigned to flash memory chips [20]. However, these collision-avoidance methods are designed only from the application viewpoint. They do not verify the actual chip collision in runtime. Though BCQ recomputes periodically the cost of read and write requests through a feedback, they do not consider the actual chip collision. Furthermore, the conflict vector method of PIQ mostly postpones the order of the large-size requests that are more likely to conflict with the other requests.

3 Dynamic Load Balanced Queuing

In this section, we first present a system model for flash-based storage with limited resources. Then, the section describes design and implementation of the proposed I/O scheduler.

3.1 Overview

Fig. 3 shows the system model of the proposed I/O scheduler, called the *Dynamic Load Balanced Queuing* (DLBQ). The scheduler handles I/O requests between multiple applications and the storage using three functions of *add*, *dispatch*, and *complete* provided by the linux kernel. An I/O request r^i consists of the starting logical address s^i, length l^i, and type (read or write) y^i. In addition, our system model directly defines the internal constraints of the flash-based storage. As shown in Fig. 3, there are a command queue with a limited size (Q) and the flash memory chips (FCs) with a limited number (P). The concurrency degree of the storage is restricted by the size of command queue or the number of flash memory chips.

To give an overview of the proposed system, Fig. 4 shows the flow chart of the modified scheduler functions of *add*, *dispatch*, and *complete*. First of all, when multiple competing applications $a_1 ... a_N$ send multiple I/O requests, the scheduler function of *add*() is called for each request. Then, DLBQ pushes a request r^i into one of two request queues of DLBQ according to the request type y^i. To minimize the interference between read and write requests, we design two queues dedicated to each request type, i.e., read and write.

Afterwards, the operating system calls the scheduler function of *dispatch*(). In the function of *dispatch*(), DLBQ finds an I/O request that can improve the utilization of flash memory chips among the backlogged requests in the activated queue. The possible storage utilization when each request is served can be computed by *balanced chip utilization* (BCU) in Section 3.4. Then, the proposed cost model is updated to reflect the storage status changed by the dispatch request.

When an I/O request is completed from the storage, the scheduler function of *complete*() is called in Fig. 4. In the function of *complete*(), the proposed cost model is adjusted by the actual completion of requests. Because the flash-based storage internally handles the complex operations

TABLE 1
Notation used

Symbols	Description
r^i	i-th request
s^i	logical page address of r^i
l^i	page length of r^i
y^i	type (read or write) of r^i
C^i	actual completion time of r^i
R^i	actual response time of r^i
I	actual processing time of page
Q	size of command queue in the storage
P	total number of flash memory chips (FCs)
c_n	index of FC accessed by the n-th page of r^i
S_k	virtual start time of the k-th FC
F_k	virtual finish time of the k-th FC
L^i	load assigned to each active FC for r^i
RF^i	virtual completion time of r^i

such as garbage collection and wear leveling, our cost model is dynamically compensated by the feedback of the storage. After that, DLBQ chooses one of two request queues to dispatch with *dynamic queue selection* (DQS) described in Section 3.3.

3.2 Cost Model

To quantitatively reduce the contention problems of read/write interference and chip collision, DLBQ presents the cost of an I/O request using a virtual time scheme. In fact, our virtual time strategy is similar with the virtual time methods of conventional schedulers [11], [28]; the existing virtual time substitutes the request length for the processing cost of request. However, on a flash-based storage, a single request can be assigned to multiple flash memory chips in parallel. In our design, we consider the parallel accesses to multiple flash chips to accurately measure the operation cost of a request. For the convenience, we summarize the notations used in this paper in Table 1.

The cost model first finds the flash chips (FCs) that are accessed by the I/O request; we refer to them as active FCs. Because the unit of the request length is equal to the striping size of 4 KB, we can get the indexes of active FCs through a modular computation of the starting address and the length of request. Thus, when a dispatch request r^i is handled in the storage, the index c_n of the active FC accessed by the n-th page is defined as follows:

$$c_n \leftarrow (s^i + n) \bmod P, \quad 0 \le n < \min(l^i, P) \tag{1}$$

where s^i is the starting page address of r^i, P is the total number of flash memory chips, and l^i is the page length of r^i. If l^i is greater than P, the upper value of n is limited to P because all flash memory chips are utilized. Otherwise, only several flash memory chips are accessed. For example, when the I/O request with the starting address of 13 and the data size of 8 KB is sent to the storage with nine chips, c_0 is four and c_1 is five because P is nine. Only two flash memory chips are accessed to process this request. On the other hand, a flash-based storage exploits the low-level resources such as die or plane within a flash memory chip to maximize the parallelism of storage [29]. In this case, P can be defined as the total number of low-level resources.

After finding the indexes of active FCs, the operation cost of each flash chip is calculated considering the request size. The operation cost is the unit time to handle the amount of request assigned to each flash chip. The operation cost $L^i_{c_n}$ of each active FC for r^i is defined as follows:

$$L^i_{c_n} \leftarrow \begin{cases} \lfloor \frac{l^i}{P} \rfloor + 1, & \text{if } (l^i \bmod P) > n \\ \lfloor \frac{l^i}{P} \rfloor, & \text{otherwise} \end{cases} \tag{2}$$

where $0 \le n < \min(l^i, P)$, c_n is the index of FC accessed by the n-th page of r^i, l^i is the page length of r^i, and P is the total number of flash memory chips. For example, let the size of an I/O request be 224 KB and P is configured as 32. Then, each operation cost for the first 24 chips is two but the operation cost for the remaining 8 chips is one.

Using equations (1) and (2), the virtual time of flash memory chip is accumulated by adding the previous virtual time and the operation cost. Therefore, the virtual start time $S_{c_n}(t)$ and virtual finish time $F_{c_n}(t)$ of each active FC are defined as follows:

$$\begin{aligned} S_{c_n}(t) &\leftarrow S_{c_n}(t-1) + L^i_{c_n} \quad \text{when completed} \\ F_{c_n}(\hat{t}) &\leftarrow F_{c_n}(\hat{t}-1) + L^i_{c_n} \quad \text{when dispatched} \end{aligned} \tag{3}$$

where $0 \le n < \min(l^i, P)$, c_n is the index of FC accessed by the n-th page of r^i, t is the event when a request completion occurs, and $\hat{t}$ is the event when a request dispatch occurs. Initially, the virtual start time $S(0)$ and virtual finish time $F(0)$ are zero. When an I/O request r^i is dispatched to the storage, the virtual finish times F_{c_n} of active FCs are incremented by the operation cost $L^i_{c_n}$. Afterwards, when the storage media responses the request completion, the virtual start times S_{c_n} of active FCs are accumulated by $L^i_{c_n}$. When completed, if the virtual start time is equal to the virtual finish time, there are no pending operations in the flash memory chip. The flash memory chip becomes inactive.

In fact, a flash-based storage internally processes various operations such as wear leveling and garbage collection. When the virtual times of specific chips are delayed due to the heavy operations, simply adding the request length is not accurately model the operation cost. To further enhance the accuracy of virtual time of the equation (3), we utilize the actual processing time of a page monitored at runtime. For that purpose, we first measure R^i which is the actual response time of r^i and calculate the actual processing time of a single page which is denoted I as follows:

$$I(t) \leftarrow \frac{R^i}{\max_n(F_{c_n}(\hat{t}) - S_{c_n}(\hat{t}))} \tag{4}$$

where $0 \le n < \min(l^i, P)$, t is the event when a request completion occurs, $\hat{t}$ is the event when a request dispatch occurs, and c_n is the index of FC accessed by the n-th page of r^i. To calculate I, the scheduler divides the actual response time by the maximum number of the accumulated operations.

To accurately track the runtime status of storage, the lagged virtual times need to be appropriately adjusted considering the actual completion of the delayed operation. Therefore, when the virtual start time of the accessing chip is behind the request completion time normalized by the

page processing time ($S_{c_n}(t) < \frac{C^i}{I(t)}$), the virtual times are adjusted to the actual time as follows:

$$\begin{aligned} S'_{c_n}(t) &\leftarrow \frac{C^i}{I(t)} \\ F'_{c_n}(t) &\leftarrow F_{c_n}(t) + (\frac{C^i}{I(t)} - S_{c_n}(t)) \end{aligned} \quad (5)$$

where t is the event when a request completion occurs, c_n is the index of FC accessed by the n-th page of r^i, C^i is the actual completion time of r^i, and I is the average processing time of a page. As shown in Fig. 4, the equation (5) is performed after the equation (3). With applying the equation (5), we can adjust the heavily lagged virtual time due to the idleness, especially when a heavy operation of garbage collection is occurred. More specifically, we can make that the virtual start time is more accurate by following the actual completion time of request and a value of virtual time is bounded to I.

The computational overhead for managing this virtual time table is $O(2P)$ in the worst case. When an I/O request is dispatched, the virtual finish times of active FCs are updated. In addition, when an I/O request is completed, the lagged virtual times are inspected for the adjustment. Though the worst overhead is proportional to the number of flash memory chips (P), the real value is limited to the constant value of relatively small size as 16, 32, or 64. We believe that our cost model can be applied for the large size SSDs with many flash memory chips. In fact, the commercial product of 16TB SSD has 512 chips [31]. The simple arithmetic computation of virtual time is negligible with a large number of flash chips because I/O performance is mostly bounded to storage performance.

3.3 Dynamic Queue Selection

On the basis of the cost model, this section solves the contention problem of the read/write interference. Like the conventional methods to reduce the interference, we also separately dispatch the read and write requests in a batch fashion. The conventional algorithms preferentially dispatch the read requests rather than the write requests [8], [21], dispatch a set of requests with the same type according to the pre-defined number of requests [22], or dispatch a set of irrelevant requests that do not access the same flash memory chips [5].

However, the previously proposed batching techniques cannot sufficiently reduce the interference between read and write requests; the access pattern of I/O requests is very complicated considering the dispatching order, pattern, or size of the requests. In contrast, our cost model can easily see the access distribution of I/O requests using the virtual time table. Thus, the method of dynamic queue selection (DQS) can be implemented and alternately schedules one of two request queues every a dynamic interval of virtual time. In addition, DQS can fairly control the activation time of read and write queues.

On the flash-based storage, the write operation of flash memory is much longer than the read operation. Also, the write operation would include additional procedures such as wear leveling and garbage collection. It would cause long response time of read requests when a write operation is processed on the same chip. To evenly distribute the activation time for dispatching between read and write queues while ensuring the fairness of delayed read requests, we first define the scheduling ratio of read queue over write queue (SR_r) as follows:

$$SR_r(t) \leftarrow \frac{TL_r(t) \times I_w(t)}{TL_w(t) \times I_r(t)} \quad (6)$$

where t is the event when a request completion occurs, TL_r is the average total workloads of requests waiting on the read queue, and I_r and I_w are the average processing time of read and write pages, respectively. The scheduling ratio of write queue SR_w is the inversion of SR_r. Basically, the scheduling ratio of read and write queues is determined by the read/write ratio of workloads generated by the host application. However, this external ratio is statically determined without considering the internal state of storage device. Thus, we additionally take into account the processing time of a page on runtime.

In general, the processing time of read page I_r is almost constant [18]. For example, the processing time of 4 KB read page takes 100 μs on Samsung SSD [30]. However, I_r can be considerably longer when the read and write requests are mixed. Because the long delay of read page is incurred by the excessive interference of write requests, we use an original value of I_r that is acquired in advance using a read-only workload. In contrast, the processing time of write page I_w is dynamically measured on runtime because it is varied by the additional operations such as wear leveling and garbage collection.

Using the equation (6), DQS can fairly control the dispatching ratio between read and write requests. However, there still remains a question on how long the selected queue is activated. If the selected queue is activated for a long time, the other queue might be severely sacrificed and therefore, the bounded latency of requests cannot be guaranteed. To solve the problem, DQS defines and uses the virtual interval NI in which a set of read or write requests utilize exclusively all flash memory chips. NI is the previous non-mixed virtual interval passed by the requests with the opposite type and the value is dynamically adjusted. By assigning a dynamic interval of NI to each queue, read or write requests can be separated adaptively from the runtime status of storage. In addition, the activation time of queue is allocated considering the current virtual finish time to exclude a mixed interval. Therefore, DQS distributes the activation time between two queues by the following virtual batch time BT_y for a request type y:

$$BT_y(t) \leftarrow \max_k F_k(t) + NI_{\neg y} \times SR_y(t) \quad (7)$$

where $0 \leq k < P$, F_k is the virtual finish time of the k-th FC, $NI_{\neg y}$ is the previous non-mixed virtual interval for the opposite type queue, and SR_y is the scheduling ratio of y-type queue. Initially, we configure the lower bound value of NI as two to ensure the pipelined execution of flash operations assigned on a chip. When the active queue is switched, the virtual batch time is recalculated by adding the maximum virtual finish time and the non-mixed virtual interval revised by the scheduling ratio.

As shown in Fig. 4, DQS switches the active queue

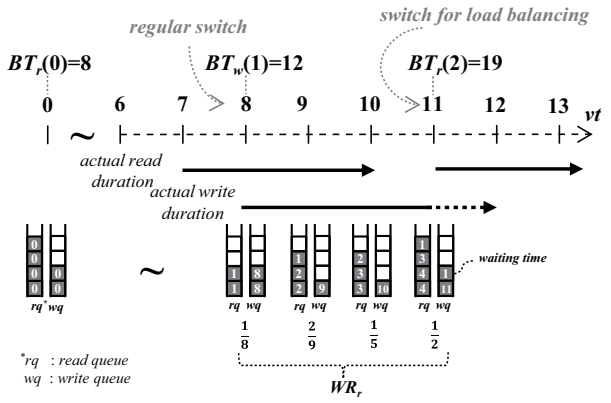


Fig. 5. Example of dynamic queue selection

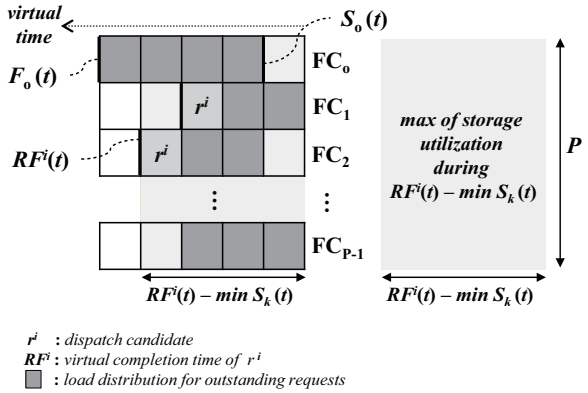


Fig. 6. Balanced chip utilization

by the following three conditions and updates the virtual batch time. Firstly, if there are no waiting requests on the active queue, the active queue is switched. Secondly, when the delay ratio of inactive write queue is greater than the scheduling ratio of active read queue, the write queue is selected. For example, if the scheduling ratio of write queue is two times of the scheduling ratio of read queue, the average waiting time of read requests cannot exceed twice of the average waiting time of write requests. Thus, the second condition for switching between read and write queues is as follows:

$$SR_r(t) < WR_w(t) \quad \text{when read queue is active} \qquad (8)$$

where WR_w is the ratio of the average waiting time of write requests over the average waiting time of read requests. When the write queue is active, the equation (8) checks that the delay ratio of inactive read queue is greater than the scheduling ratio of write queue ($SR_w < WR_r$). Lastly, when the minimum virtual start time is greater than equal to the virtual batch time, the active queue is switched. The third condition for switching between read and write queues is as follows:

$$\min S_k(t) \geq BT(t-1) \qquad (9)$$

where $0 \leq k < P$, S_k is the virtual start time of the k-th FC, and BT is the virtual batch time. According to the equation (3), the virtual start time is updated when a flash memory chip is actually used. In this regard, the minimum virtual start time is increased only if all flash chips are used at least once. Thus, to guarantee that a set of read or write requests use exclusively all flash chips, DQS compares the minimum virtual start time to the virtual batch time.

Fig. 5 shows an example of the proposed dynamic queue selection (DQS). The gray box on each queue represents the waiting requests and the page length of the requests is all one. When the requests are served from the queue, the gray box is changed to the white box. The number denoted in the gray box indicates the time passed since the corresponding request is enqueued. To easily explain DQS, let us assume that the average read/write ratio of workloads is 2:1. In addition, the processing times of read and write page are configured as 100 μs and 200 μs, respectively. Thus, the scheduling ratio of read queue SR_r is four, whereas the scheduling ratio of write queue SR_w is 1/4. Further assume that initially there are four read requests and two write requests and one unit of virtual time as 100 μs.

Initially, the read queue is activated and the virtual batch time $BT_r(0)$ is set to eight according to the equation (7); SR_r is four and the non-mixed virtual interval NI is two. The read requests are continuously dispatched/completed from the read queue, whereas the write requests are waiting. Later, when the minimum virtual start time becomes eight, the write queue is activated according to the equation (9). At this point, two read requests with the waiting time of one are held in the read queue. In addition, assuming that a read request with the virtual finish time of ten exists in the storage. Thus, the next virtual batch time $BT_w(1)$ is set to the sum of the maximum virtual finish time of ten and NI of two. Though NI passed by the read requests is eight, NI is decreased into two due to SR_w. When the virtual time is 11, the delay ratio of read queue WR_r exceeds the scheduling ratio of the write queue SR_w. According to the equation (8), the active queue is changed to the read queue and the next virtual batch time $BT_r(2)$ is recalculated.

3.4 Balanced Chip Utilization

After choosing the target queue with DQS, the second algorithm of DLBQ, called Balanced Chip Utilization (BCU), reorders I/O requests in the queue to alleviate the contention problem of chip collision. The reorder policy of BCU is to evenly distribute workload to flash memory chips while maintaining a bounded latency.

To achieve well-balanced utilization of storage, we should first acquire the total load of flash memory chips in runtime. According to the equation (3) and (5), the load amount of each flash chip is calculated by the difference between the virtual finish time and the virtual start time. In addition, the total load amount includes the load amount of a dispatch candidate that is waiting in the queue. Therefore, in sum, the total load amount for a dispatch candidate r^i is defined as follows:

$$TC(t) \quad \leftarrow \quad \sum_{k=0}^{P-1} \big(\min(RF^i(t), F_k(t)) - S_k(t) \big) \qquad (10)$$

where k is the index of flash memory chip, P is the total number of flash memory chips, RF^i is the virtual completion time of r^i, F_k is the virtual finish time of the k-th FC, and S_k

TABLE 2
Workload characteristics

Workloads	Patterns	I/O Sizes	R/W Ratios	Number of threads
Micro Benchmark	Random	4 KB–32 KB	1:1	32–256
Web Server	Random	0.5 KB–64 KB (AVG: 11 KB)	9:1, 1:9	100
File Server	Random	4 KB–512 KB (AVG: 64 KB)	9:1, 1:9	100
Database	Random	8 KB–64 KB (AVG: 22 KB)	2:1, 1:2	200
Mail Server	Sequential	16 KB	4:1, 1:1, 1:4	130

is the virtual start time of the k-th FC. The virtual completion time RF^i is determined as the max value among the virtual finish times of the flash chips accessed by the dispatch candidate r^i. Because a single request does not access all flash memory chips always, the virtual finish times of some chips can be greater than the virtual completion time of the request. Thus, if the virtual finish time F_k is greater than the virtual completion time RF^i, the load amount is calculated only up to RF^i. The reason for limiting the upper bound of the total load amount is to find an I/O request that maximizes the utilization in the expected finish time of request.

Now we can calculate the utilization ratio and choose a dispatch request based on the ratio. As shown in Fig. 6, the utilization is defined as a ratio of the total amount of load over the maximum load amount for the completion of a dispatch candidate. Therefore, the utilization for each dispatch candidate r^i is estimated as follows:

$$U(t) \quad \leftarrow \quad \frac{TC(t)}{P \times (RF^i(t) - \min S_k(t))} \tag{11}$$

where $0 \leq k \leq P$-1, $TC(t)$ is the total load amount of flash memory chips in runtime, P is the total number of flash memory chips, S_k is the virtual start time of the k-th FC, and RF^i is the virtual completion time of r^i. As shown in Fig. 4, the utilization is investigated for Q-requests which are pending in the request queue; to bound the latency of requests, the number of dispatch candidates is limited to the size of the command queue (Q). Among them, a request with the maximum utilization is selected and chosen from the request queue and transmitted to the storage; any request can be selected among the backlogged requests and therefore, BCU employs the request queue of the linked-list structure.

When a new request is issued from the host, the utilization of the request is examined together with the waiting requests issued in advance. If the utilization of the new request is greater than that of the waiting requests, the waiting requests cannot be dispatched. When the waiting requests are not dispatched repeatedly, their latencies increase significantly. To bound the worst-case latency of requests, BCU compensates the waiting requests for the deficit time $D(t)$; in fact, this methodology is inspired by the deficit time of DRR [32]. Therefore, the storage utilization of a dispatch candidate r^i is finally defined as follows:

$$U(t) \quad \leftarrow \quad \frac{TC(t) + D(t-1)}{P \times (RF^i(t) - \min S_k(t))} \tag{12}$$

where $D(t$-1) is the deficit time of the dispatch candidate. Initially the deficit time of request is zero. After a dispatch request is determined, the deficit times of the requests enqueued earlier than the dispatch request are updated. For each request, the deficit time is accumulated by adding the previous deficit time and the length of dispatch request. In the next dispatch, the utilization of the waiting requests increases by $D(t$-1), and hence their dispatching chance also increases.

As explained before, the number and size of the outstanding I/O requests is finite due to the limited concurrency of storage. Accordingly, the load interval ($RF^i(t)$ - $\min S_k(t)$) cannot be larger than a certain value. In contrast, when some requests are staying in the request queue persistently, their deficit times $D(t)$ steadily increase. If the deficit time of a waiting request is greater than the total load amount, the waiting request is dispatched unconditionally.

In overall, the computational overhead of DLBQ is $O(2P + Q \times P + Q)$ in the worst case, where P is the total number of flash memory chips and Q is the size of command queue. As mentioned in Section 3.2, the overhead for managing the virtual times is $O(2P)$. In addition, the overhead to search a dispatch request is $O(Q \times P)$. If the Q-th request is dispatched from the request queue, the computation of the deficit times needs the overhead of $O(Q)$. On the other hand, the space complexity of DLBQ is $O(2P+2Q)$. Two virtual times are needed per a chip. In addition, the maximum load of the accessing chips and the deficit time are handled for each request.

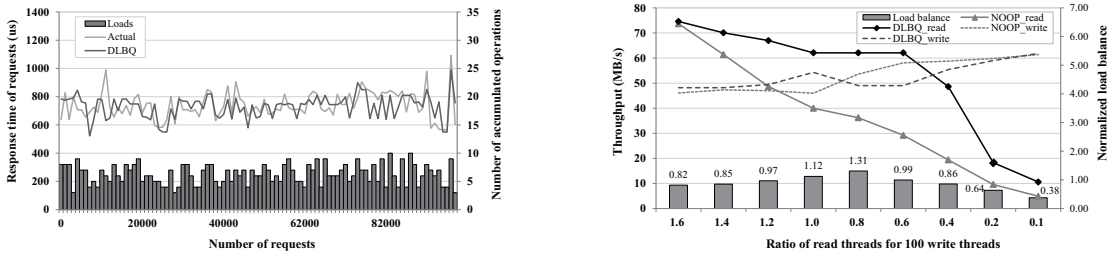
4 Experiments

In this section, we evaluate the proposed scheduler with various workloads. To verify the effectiveness of DLBQ, we first validate our cost model. Then, we evaluate the micro-benchmark and server benchmarks with various data sets. In addition, we show the performance improvement of DLBQ compared with the previous I/O schedulers.

4.1 Experimental Setup

All of the experiments presented in this section are conducted on a Intel(R) Core(TM) i5-3330 (3.0 GHz, four cores) and 4 GB of RAM. To measure the enhancement due to DLBQ for flash-based storage, we use two Samsung SSDs (PM850 Pro) [30] of 128 GB and 256 GB attached to the storage interface of SATA 6G [23]. Because the number of flash memory chips is differently configured depending on the storage capacity, we test two SSDs of different capacities; the reason for using the SSDs with the same industry is because the internal algorithms or policy of SSDs considerably differs according to industry. In DLBQ, the total flash memory chips of 128 GB and 256 GB storage (P) are set to 16

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2636840, IEEE Transactions on Computers



(a) Request response time according to the number of accumulated operations

(b) Throughput variation according to the different read/write ratios of workload

Fig. 7. Cost model validation

and 32, respectively. Also, the size of command queue (Q) is set to 32. DLBQ is implemented on the GNU/Linux operating system with kernel version 3.5.0. DLBQ is compared to the previous I/O schedulers of NOOP [33], FlashFQ [28], ParDispatcher [22], and PIQ [5].

The main purpose of our experiments is to validate whether the throughput and latency of the flash-based storage is improved by using the DLBQ scheduler. For this purpose, we exploit Flexible I/O generator (Fio) that can produce raw I/Os [34]. To produce asynchronous I/Os, the ioengine of Fio is configured as Linux Native AIO (libaio). In addition, Direct I/O is used to bypass the data buffering within the file system. To reduce the performance dependency between experiments, the experiment always precedes the SSD TRIM command [23]; the frequency of garbage collections can decrease because TRIM command informs SSDs about the invalid blocks, which are no longer used in the storage device. Each measurement performs 10 million requests.

Table 2 summarizes the characteristics of various workloads generated by Fio. Basically, the I/O size, read/write ratio, patterns, and threads of workloads are differently configured; the I/O depth of all threads is limited to one. The workloads are categorized in two types of micro-benchmark and server benchmark.

4.2 Cost Model Validation

In order to validate our cost model, we first evaluate the accuracy of our delay estimation using equation (3) and (5). Fig. 7(a) plots the response time of requests for the web server workload on the 128 GB SSD. The actual response time of requests varies between 600 μs and 1000 μs. As shown in Fig. 7(a), we can observe that the response time varies according to the number of the accumulated operations represented in the bar graph. Similarly, our cost model (DLBQ) well follows the actual response time depending on the number of operations. The error rate of DLBQ is 8 % on average.

From the results, we demonstrate that our cost model is accurate for the read-intensive workload such as web server. However, when the write-intensive workload such as file server causes an operation of garbage collection, it might be a little difficult to provide an accurate modeling; the equation (3) is basically designed by the external loads produced by host application and therefore it is difficult to predict the internal loads produced by the garbage collection in advance. In fact, a major issue of storage performance for the write-oriented workload is to prevent the significant delay of read requests from the interference of write requests. To guarantee the fairness of read performance, this paper designs the dynamic queue selection (DQS). Accordingly, the second experiment of cost model validation verifies whether DQS can fairly dispatch the read requests from the superfluous interference of write requests.

To generate the scenario that frequently carries out the heavy operations of wear leveling and garbage collection, 4 KB random workloads has been requested after fully writing 128 GB SSD. We have varied the number of read threads while maintaining 100 write threads to change the read/write workload ratio. Fig. 7(b) shows that the read throughput of DLBQ highly outperforms that of NOOP for all read/write ratios, while guaranteeing a constant write throughput. The read throughput of DLBQ is greater than the write throughput until the read ratio of workload is 0.4. In contrast, all read throughputs of NOOP significantly decrease as the read ratio of workload decreases. Even if the read ratio of workload (1.0) is equal to the write ratio, the read throughput of NOOP is less than the write throughput.

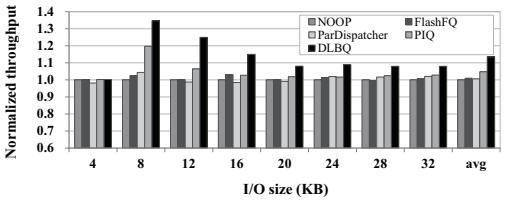
Meanwhile, we additionally validate a load balance metric that is measured by the deviation between read and write throughput; as the deviation is greater, the load balance is lower. The bar graph of Fig. 7(b) shows that the load balance of DLBQ is normalized by the load balance of NOOP. As shown in Fig. 7(b), the normalized load balance of DBLQ is mostly smaller than one as 0.88 on average. Especially, DLBQ considerably harmonizes the loads between read and write queues when the read proportion of workload becomes smaller.

4.3 Micro Benchmark

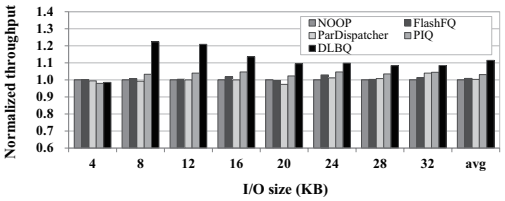
The micro-benchmark is aimed at evaluating the storage throughput for five schedulers when the workload parameters, i.e., I/O size and the number of threads, are statically varied. In the benchmark, the other parameters of read/write ratio and I/O pattern are fixed 1:1 and a random pattern, respectively.

Fig. 8 shows the throughput variation according to the request size. This experiment examines how well an I/O scheduler mitigates the contention between I/O requests. Because the contention is increased according to the number of accessing flash chips, the request size increases from 4 KB to 32 KB by 4 KB unit. The number of threads is set to 128.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2636840, IEEE Transactions on Computers

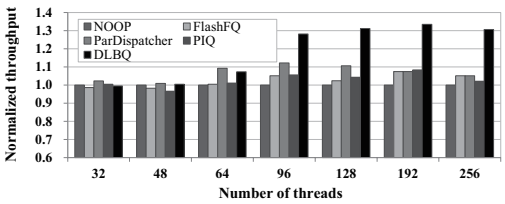


(a) Normalized throughput on 128 GB SSD

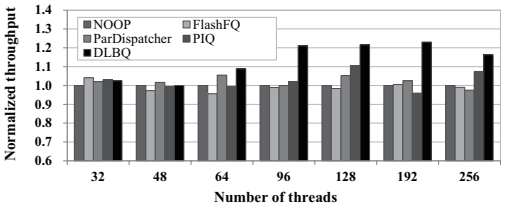


(b) Normalized throughput on 256 GB SSD

Fig. 8. Throughput comparison according to I/O size



(a) Normalized throughput on 128 GB SSD



(b) Normalized throughput on 256 GB SSD

Fig. 9. Throughput comparison according to the number of threads

As shown in Fig. 8, the throughput of DLBQ outperforms the throughput of the other schedulers for all I/O sizes; the throughput of all schedulers is normalized by the throughput of NOOP. On average, the throughput of DLBQ is 10 % greater than the throughput of NOOP on both SSDs. Especially, the throughput enhancement is highly greater at the requests with medium size (8 KB∼16 KB). It means that the medium-sized requests frequently incur the contention problems of storage. Meanwhile, the throughput for 4 KB I/O does not improve even though an I/O scheduler is exchanged. It is because the requests with a single page (4 KB) can be mostly avoided by preferentially serving the other idle chips. In addition, the throughput improvement of large-sized requests is smaller because the requests fully utilize all flash memory chips.

The second experiment of micro-benchmark validates the throughput variation according to the number of threads. This experiment checks how well an I/O scheduler leverages the increasing number of backlogged I/O requests. For this experiment, the total number of threads increases from 32 to 256 and the request size is fixed 8 KB.

Fig. 9 shows that the throughput of DLBQ greatly exceeds the throughput of the other schedulers. Up to 64 threads, the throughput of DBLQ is similar with the other schedulers because the number of backlogged requests is not enough to reorder. Since the storage handles internally 32 outstanding requests, the remaining requests are only backlogged in the queue of scheduler. However, the throughput of DLBQ remarkably outperforms the other beyond 96 threads. The throughput of DLBQ is 30 % greater than the throughput of the other schedulers. The key intention behind the increase of the number of threads is that the opportunity of raising the balance of the distributed load of flash memory chips increases. Thus, the throughput of DLBQ can increase as the number of threads increases. In contrast, the throughputs of the other schedulers are not improved much even though the number of threads increases.

4.4 Server Benchmark

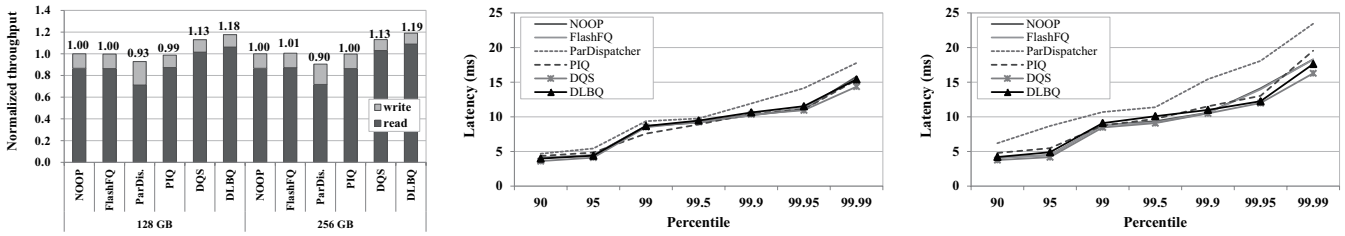
The server benchmark evaluates four common server workloads of web server, file server, database, and mail server. These server workloads are referenced by the work of Sehgal et al. [24]. Like the micro-benchmark, DLBQ is compared to the four schedulers of NOOP, FlashFQ, ParDispatcher, and PIQ. In addition, the performance result of dynamic queue selection (DQS) is added to verify the performance impact of two algorithms, DQS and BCU, in DLBQ.

The experiments illustrate both the throughput and latency in detail. The throughput results are normalized by the throughput of the NOOP scheduler. In addition, the throughput results distinguish between read and write throughput. Meanwhile, the latency evaluation shows the latency percentile of 90th or more because the worst-case latency can severely increase due to the reordering policy of schedulers; in fact, the most server applications want to minimize tail latencies, which correspond to the worst user experience [35].

4.4.1 Evaluation of Web Server Workload

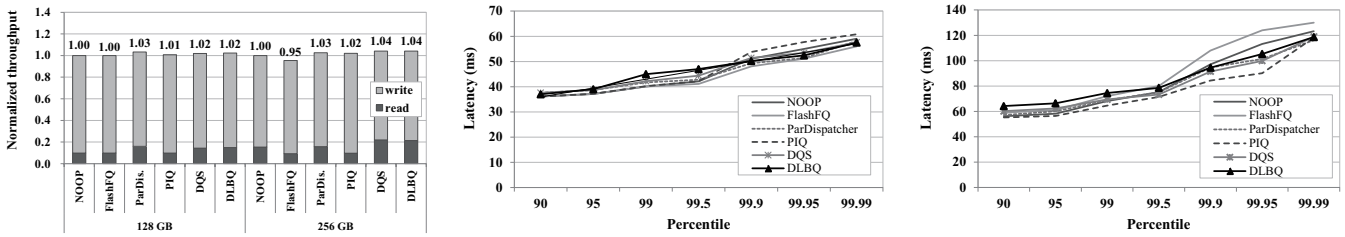
The web server workload emulates the web applications which frequently perform fast lookup for small-size files such as Web page or Web log. To generate such a read-intensive workload with small size, the workload is configured as the read/write ratio of 9:1 and the request length of mostly 4 KB or less. This workload is produced by 100 threads with a random pattern.

Fig. 10(a) shows that the normalized throughput of DLBQ outperforms the throughput of the other schedulers for the web server workload. Especially, only with the dynamic queue selection (DQS) method of DLBQ, the throughput considerably increases by 13 % on the 128 GB and 256 GB SSD, when compared to the throughput of NOOP. It means that the read requests are easily delayed by the write requests. Accordingly, if the read and write requests are sufficiently decoupled, the read throughput can largely increase. Furthermore, when BCU is additionally used, the



(a) Throughput (b) Latency on 128 GB SSD (c) Latency on 256 GB SSD

Fig. 10. Throughput and latency for web server workload



(a) Throughput (b) Latency on 128 GB SSD (c) Latency on 256 GB SSD

Fig. 11. Throughput and latency for file server workload

throughput more increases by 5 % on the 128 GB and 6 % on the 256 GB SSD. As denoted in Table 2, the average request size of the web server workload is 11 KB. Because a single request accesses only two or three chips on the storage, it results in the idleness of chips when different requests access the same chip. Thus, BCU can preferentially dispatch the requests that can increase the storage utilization.

In contrast, the throughput of ParDispatcher [22] is significantly worse than the throughput of the other schedulers. The throughput is declined by 7 % on the 128 GB SSD and 10 % on the 256 GB SSD when compared to the throughput of NOOP. It is because the batch size of ParDispatcher is statically predefined at compile time; in ParDispatcher, the numbers of read and write requests belonging to a batch are set to 32 and 16, respectively. This static batching method transmits I/O requests depending on the internally fixed ratio, irrespective of the external ratios of workloads. For this web server workload, the ratio of read and write bandwidth of ParDispatcher is nearly 4:1 (read: 241 MB/s, write: 63 MB/s) on the 256 GB SSD, even though the external ratio is 9:1. In order to match with the internal ratio of 2:1, the write throughput slightly increases but the read throughput considerably decreases. The measurement shows that the static batching method decreases the storage throughput more when the predefined batch ratio does not coincide with the external ratio of workloads.

On the other hand, Fig. 10(b) and Fig. 10(c) show the long-tail latencies for the web server workload. The latencies of all schedulers are similar to each other except for the latency of ParDispatcher. Because the average size of web server workload is small in overall, the latencies of most schedulers are the same. However, similar to the throughput results, the static batching method of ParDispatcher gives an adverse effect on the long-tail latencies.

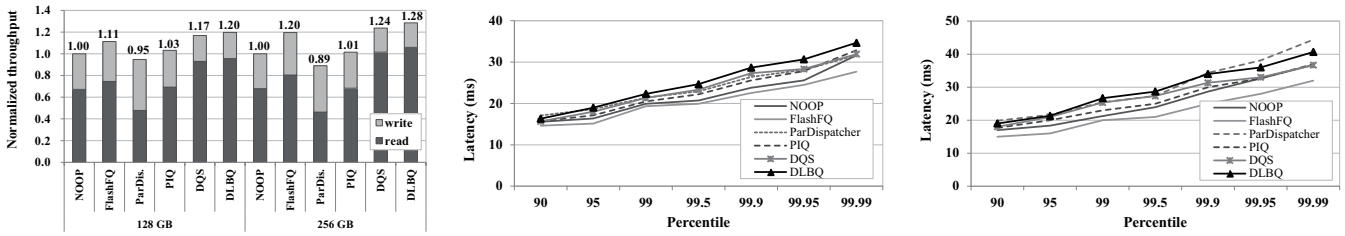
4.4.2 Evaluation of File Server Workload

The file server workload emulates the file server applications that provide the storing and sharing of large-size files to users. The length of I/O requests is proportional to the file cluster size of 4 KB. To generate such a write-intensive workload, the workload is configured as the read/write ratio of 1:9 and is composed of I/O requests with mostly 32 KB or more. Also, the workload has a random pattern to access files with various sizes.

Fig. 11(a) shows the throughput results for the file server workload. In contrast to the web server results, the throughput of all schedulers are within 4 % difference compared to NOOP. There is not much performance variation due to the effectiveness of I/O schedulers. As described in Table 2, the average request size of the file server workload is 64 KB. It means that a single request accesses 16 flash memory chips at one time, and so most of flash chips are always activated. Therefore, the throughput is not improved largely even if the dispatching order of requests is changed.

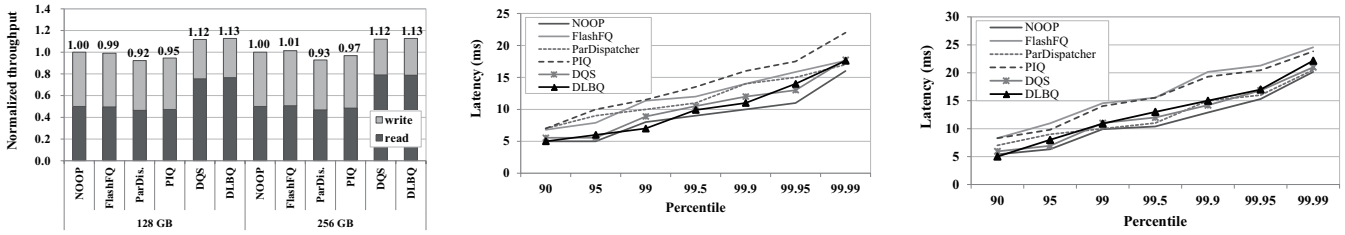
However, this write-oriented workload is hard to ensure the fairness of read requests. The response time of read requests is longer due to the heavy operations associated with write requests. Thus, we consider the actual operation time of read/write page to prevent the long latency of read requests. As shown in Fig. 11(a), the read throughput of DLBQ improves by 4 % on the 128 GB and 256 GB SSD.

Meanwhile, when the request size is large, I/O reordering is not effective as shown in Fig. 11(a). Nevertheless, if reordering is aggressively performed, the latency of requests is considerably longer. Fortunately, such reordering is limited by the latency restrain strategies of DLBQ such as the load balancing between read/write queues and the loss time of waiting requests. As shown in Fig. 11(b) and Fig. 11(c), the latency of DLBQ is similar with the latency of the other schedulers.



(a) Throughput (b) Latency on 128 GB SSD (c) Latency on 256 GB SSD

Fig. 12. Throughput and latency for database workload



(a) Throughput (b) Latency on 128 GB SSD (c) Latency on 256 GB SSD

Fig. 13. Throughput and latency for mail server workload

4.4.3 Evaluation of Database Workload

The database workload emulates a specific systems, called *online transaction processing* (OLTP), which facilitates and manages transaction-oriented applications such as order, finance, or retail. On the typical database engine, the OLTP workloads tend to select a small number of records in a table at a time. Thus, the storage receives the data of medium size which is uniformly distributed from 8 KB to 64 KB. For this experiment, 200 threads produce the I/O requests with a random pattern and the ratio of read/write threads is 2:1.

Fig. 12(a) demonstrates that the throughput of DLBQ is better than the throughput of the other schedulers for the database workload. The throughput of DLBQ increases by 20 % on the 128 GB and 28 % on the 256 GB SSD. Similar to the throughput results of web server workload, the throughput of DQS highly increases by 17 % on the 128 GB and 24 % on the 256 GB SSD. However, one thing to note is that the throughput of DQS more increases by 7 % when the storage capacity increases. As the storage capacity increases, the number of flash memory chips increases. It means that the read and write requests are frequently interfered to each other because the number of accessing chips increases. Accordingly, the number of I/O requests belonging to a batch must be varied according to the storage capacity. Meanwhile, our cost model is basically constructed according to the total number of flash memory chips (P). Thus, DQS based on the cost model can well decouple the read and write requests even though the storage capacity is changed.

4.4.4 Evaluation of Mail Server Workload

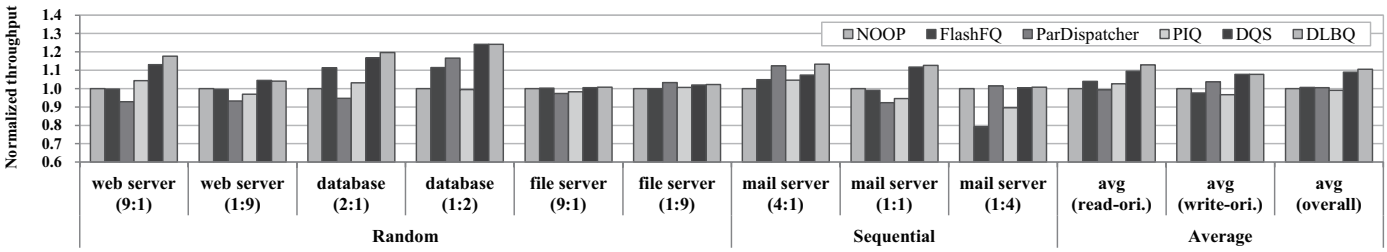
The mail sever workload emulates the internet softwares such as electronic mail, netnews, or web-based commerce. Unlike the above three workloads, the mail server workload is sequentially appended by 130 threads. Because the size of sequential requests is typically fixed, the length of I/O request is set to 16 KB. The ratio of read and write threads is configured as 1:1.

Fig. 13(a) shows that the throughput of DLBQ is highest for the mail server workload. The throughput of DLBQ increases by 12 % on the 128 GB and 256 GB SSD. However, the throughput gain from the method of BCU increasingly decreases as the write ratio of workload increases. The throughput gains of web server workload, database workload, and mail server workload are 6 %, 3 %, and 1 %, respectively. Because recent SSDs are mostly equipped with a write buffer internally, the data of write requests are first piled up in the write buffer. Thus, the reordering method of BCU is effective only for read throughput and the performance impact of BCU is different depending on the read ratio of workloads.

On the other hand, Fig. 13(b) and Fig. 13(c) show that the long-tail latencies of DLBQ are slightly higher than the latencies of the other schedulers. This latency trend similarly appears to the database workload in Fig. 12(b) and Fig. 12(c). The cause of latency increase is that DLBQ more assigns the read proportion of workload according to the actual processing time of a single page. It increases the response time of write requests and thus the long-tail latency becomes longer. However, the increase rate of latency is bounded to 10 % because long waiting requests are preferentially served.

4.4.5 Overall Throughput Evaluation of Server Workloads

Each of the above server workloads shows the performance results for the only one case of read and write ratio. However, the performance of flash-based storage is considerably affected by the read/write ratio [36]. Because the mixed requests cause various internal operations such as read prefetch and the flush of dirty data, their competition results in diverse performance degradation. Thus, to extensively validate the throughput enhancement of DLBQ, we change



(a) 128 GB SSD

(b) 256 GB SSD

Fig. 14. Overall throughput for server workloads

symmetrically the read/write ratios of the above server workloads and evaluate additionally the workloads. This section also shows the average throughput of read-oriented, write-oriented, and overall workloads. In addition, the throughput of random workloads and sequential workloads are separated.

Fig. 14 shows the throughput results for all datasets of server workloads. Looking at the read-oriented workloads, the throughput of DLBQ is highly greater than the throughput of the other schedulers. On average, the throughput is improved by 13 % on the 128 GB SSD and 17 % on the 256 GB SSD. According to the throughput results of DQS, we can first observe that the sufficient decoupling between read and write requests is considerably effective for achieving higher throughput. Further, the I/O reordering of BCU is effective when the read ratio of workloads are relatively higher. When the dispatch order of I/O requests are re-assigned to mitigate the chip collisions between I/O requests, the storage utilization can be more improved. Owing to BCU, the throughput of DLBQ more increases by 4 % on average.

Meanwhile, for the write-oriented workloads, the average throughput of DLBQ increases by 8 % on the 128 GB SSD and 14 % on the 256 GB SSD. Especially, the results on the 1:2 database workload considerably improve the throughput by 24 % on the 128 GB SSD and 38 % on the 256 GB SSD. In fact, the results of ParDispatcher are also very competitive; this is because both approaches aim to prevent the unfair scheduling of read operations blocked by the long write operations. However, as mentioned before, the static batching method of ParDispatcher significantly degrades the performance with the read-oriented workloads. The proposed DLBQ can successfully improve the performance drop caused by the read/write interference for diverse workload sets.

In overall, DLBQ achieves a 11% of performance improvement on the 128 GB SSD and 15 % on the 256 GB SSD for four common server workloads. Especially, when the workloads are relatively small sizes and random pattern, the storage throughput is highly improved. In addition, DLBQ effectively improves the throughput even if the number of flash memory chips increases depending on the storage capacity. Conclusively, DLBQ accomplishes high throughput of storage for various system environments.

5 Conclusions

We discovered that the previous works on I/O scheduling are not optimized when the performance of flash-based storage greatly varies according to the internal constraints of the storage, such as static striping and limited concurrency. In particular, when workloads result in *contention* in which requested data are handled only by a subset of the parallel resources within the storage, the performance is considerably degraded. To solve the contention problems, we propose the *Dynamic Load Balanced Queuing* (DLBQ) which schedules I/O requests to maximize the storage utilization while limiting the long-tail latency. For that purpose, we have introduced the cost model which represents the load assigned to flash memory chips using a virtual time method. On the basis of the cost model, the strategy of *dynamic queue selection* (DQS) prevents the mixing of read/write requests within the storage device. Also, the method of *balanced chip utilization* (BCU) maintains the load asymmetrically distributed to flash memory chips as balanced as possible.

We have evaluated the throughput and latency of DLBQ for various workloads. By varying the parameters of the workloads such as the I/O patterns, read/write ratios, and sizes, we compared the throughput and latency versus the previously proposed I/O schedulers for the flash-based storage. Consequently, we have shown that DLBQ delivers a significantly higher throughput than the other scheduling algorithms while maintaining the worst-case latency.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance." in *USENIX Annual Technical Conference*, 2008, pp. 57–70.

[2] C. Dirik and B. Jacob, "The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 279–289.

[3] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and Exploiting the Multilevel Parallelism Inside SSDs for Improved Performance and Endurance," *Computers, IEEE Transactions on*, vol. 62, no. 6, pp. 1141–1155, 2013.

[4] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[5] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H. Sha, "Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives," in *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*. IEEE, 2014, pp. 1–11.

[6] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 524–535.

[7] M. Jung, E. H. Wilson III, and M. Kandemir, "Physically addressed queueing (PAQ): improving parallelism in solid state disks," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 404–415.

[8] Q. Zhang, D. Feng, F. Wang, and Y. Xie, "An Efficient, QoS-Aware I/O Scheduler for Solid State Drive," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 2013, pp. 1408–1415.

[9] S.-y. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based SSDs," *Computer Architecture Letters*, vol. 9, no. 1, pp. 9–12, 2010.

[10] M. Jung and M. Kandemir, "Revisiting widely held SSD expectations and rethinking system-level implications," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1. ACM, 2013, pp. 203–216.

[11] W. Jin, J. S. Chase, and J. Kaur, "Interposed Proportional Sharing for a Storage Service Utility," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1. ACM, 2004, pp. 37–48.

[12] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, "Storage Performance Virtualization via Throughput and Latency Control," *ACM Transactions on Storage (TOS)*, vol. 2, no. 3, pp. 283–308, 2006.

[13] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance Insulation for Shared Storage Servers." in *FAST*, vol. 7, 2007, pp. 5–5.

[14] A. Gulati, I. Ahmad, C. A. Waldspurger *et al.*, "PARDA: Proportional Allocation of Resources for Distributed Storage Access." in *FAST*, vol. 9, 2009, pp. 85–98.

[15] P. Valente and F. Checconi, "High Throughput Disk Scheduling with Fair Bandwidth Distribution," *Computers, IEEE Transactions on*, vol. 59, no. 9, pp. 1172–1186, 2010.

[16] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin, "Maestro: Quality-of-Service in Large Disk Arrays," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 245–254.

[17] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: Enabling High-level SLOs on Shared Storage Systems," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 14.

[18] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *Proceedings of the seventh ACM international conference on Embedded software*. ACM, 2009, pp. 295–304.

[19] M. P. DUNN, "A new I/O scheduler for solid state devices," Ph.D. dissertation, Texas A&M University, 2009.

[20] M. H. Jo and W. W. Ro, "Contention-Free Fair Queuing for High-Speed Storage with RAID-0 Architecture," in *High Performance Computing and Communications (HPCC), 2015 IEEE 17th International Conference on*. IEEE, 2015, pp. 175–183.

[21] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler." in *FAST*, 2012, p. 13.

[22] H. Wang, P. Huang, S. He, K. Zhou, C. Li, and X. He, "A novel I/O scheduler for SSD with improved performance and lifetime," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 2013, pp. 1–5.

[23] Serial ATA International Organizations, "Serial ATA Revision 3.1," 2011.

[24] P. Sehgal, V. Tarasov, and E. Zadok, "Evaluating performance and energy in file system server workloads." in *FAST*, 2010, pp. 253–266.

[25] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choi, S. Yoon, and J. Cha, "VSSIM: Virtual Machine based SSD Simulator," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 2013, pp. 1–14.

[26] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4. ACM, 1989, pp. 1–12.

[27] A. K. Parekh and R. G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," *IEEE/ACM Transactions on Networking (ToN)*, vol. 1, no. 3, pp. 344–357, 1993.

[28] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs." in *USENIX Annual Technical Conference*, 2013, pp. 67–78.

[29] M. Jung and M. T. Kandemir, "An Evaluation of Different Page Allocation Strategies on High-Speed SSDs." in *HotStorage*, 2012.

[30] "Samsung PM850 Pro," *URL http://www.samsung.com/global/business/semiconductor/minisite/SSD/downloads/document/Samsung_SSD_850_PRO_Data_Sheet_rev_2_0.pdf*, 2015.

[31] "Samsung PM1633a," *URL http://www.samsung.com/semiconductor/global/file/insight/2016/06/PM1633a-flyer-0.pdf*, 2016.

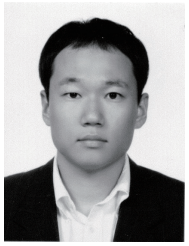
[32] M. Shreedhar and G. Varghese, "Efficient Fair Queuing using Deficit Round Robin," *Networking, IEEE/ACM Transactions on*, vol. 4, no. 3, pp. 375–385, 1996.

[33] J. Axboe, "Linux block IOpresent and future," in *Ottawa Linux Symp*. Citeseer, 2004, pp. 51–61.

[34] ——, "Fio-flexible io tester," *URL http://freecode. com/project s/fio*, 2008.

[35] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 4.

[36] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1. ACM, 2009, pp. 181–192.



Myung Hyun Jo He received his B.S. and M.S. degree in Computer Science and Engineering from Hanyang University, Ansan, Korea, in 2004 and 2006. He has been worked as a senior engineer in the Software Development Team, Memory Division, Samsung Electronics. He is also currently pursuing his Ph.D. degree in the School of Electrical and Electronic Engineering, Yonsei University, and is with the Embedded Systems and Computer Architecture Lab. His current research interests are I/O scheduling and load balancing using SSD and SSD microarchitecture design.

Won Woo Ro He received his B.S. degree in Electrical Engineering from Yonsei University, Seoul, Korea, in 1996. He received his M.S. and Ph.D. degrees in Electrical Engineering from the University of Southern California in 1999 and 2004, respectively. He worked as a research scientist in the Electrical Engineering and Computer Science Department at the University of California, Irvine. He currently works as an Assistant Professor in the School of Electrical and Electronic Engineering at Yonsei University. Prior to joining Yonsei University, he worked as an Assistant Professor in the Department of Electrical and Computer Engineering at the California State University, Northridge. His industry experience also includes a college internship at Apple Computer Inc. and a contract software engineer with ARM Inc. His current research interests are high-performance microprocessor design, compiler optimization, and embedded system designs.