JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 17, No. 2, June 2023, pp. 80-92

Accurate Calibration and Scalable Bandwidth Sharing of Multi-Queue SSDs

Hyeongseok Kang
Department of Information Communication, Soongsil University, Seoul, Korea
hseokkang@gmail.com

Kanghee Kim*
School of Artificial Intelligence Convergence, Soongsil University, Seoul, Korea
khkim@ssu.ac.kr

Abstract

The emerging multi-queue solid state drives (SSDs) impose two challenges on I/O scheduling in the host operating system. First, the I/O scheduler should give a scalable performance in the number of processor cores to exploit the massive parallelism within the SSD. Second, it should provide performance isolation between the cores so that each core can schedule application I/O streams with a reserved bandwidth share. To cope with these challenges, we propose a novel I/O scheduler called mqFlashFQ. In mqFlashFQ, for every core to make a scheduling decision in parallel, we use a randomization technique to decentralize the existing FlashFQ algorithm, consequently, significantly reducing the inter-core synchronization overheads. Moreover, to provide a fair bandwidth share on a per-core basis, we present an accurate calibration method that determines the cost of each I/O request in terms of its direction and size. This method is distinguished in that it enables to provide a minimum bandwidth guarantee to each core with no garbage collection. Through our experiments with non-volatile memory express (NVMe) SSD products, we demonstrate that the proposed mqFlashFQ and calibration method give a scalable performance and a fair share of the bandwidth to each core for various I/O workloads.

Category: Real-Time Systems

Keywords: Multi-queue SSDs; I/O scheduling; Fair queuing; Randomization; Request cost calibration

I. INTRODUCTION

Solid state drives (SSDs) are now widely used in servers with high bandwidth requirements such as databases, service clouds, and virtual machine servers [1, 2]. The traditional SSDs use the Advanced Host Controller Interface (AHCI) [3] on Serial ATA (SATA) bus, which allows a single I/O queue to interface between the host operating system (OS) and the SSD controller. As more flash chips are embedded within an SSD for higher bandwidth, the single queue quickly becomes a bottleneck between multiple processor cores that try to exploit the bandwidth with multiple I/O threads. To overcome this limitation, a new storage interface called non-volatile memory express (NVMe) [4] has been proposed on the peripheral component interconnect express (PCIe) bus.

Open Access **http://dx.doi.org/10.5626/JCSE.2023.17.2.80** **http://jcse.kiise.org**

Received 28 February 2023; **Accepted** 05 June 2023
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

This new interface allows multiple I/O queues between the host OS and the SSD controller, thus enabling multiple cores to dispatch I/O requests in parallel through their respective queues. As a result, the theoretical limit on the SSD throughput has been increased to about 10 times, i.e., from several hundred megabytes to several gigabytes per second (the maximum transfer rate of SATA version 3 is 768 MB/s while that of PCIe version 3 is about 8 GB/s assuming that eight lanes of the PCIe bus are used for an NVMe SSD, each lane having a maximum transfer rate of about 1 GB/s).

However, the emerging multi-queue SSDs impose two challenges on I/O scheduling in the host OS. First, the I/O scheduler should give a scalable performance in the number of processor cores to exploit such a high bandwidth of the SSD. For example, for a server supporting several virtual machines, to fully utilize the potential bandwidth, the scheduler needs to allow each core to dispatch I/O requests in parallel. However, if the parallel dispatch requires inter-core synchronizations in accessing global variables associated with a system-level QoS metric, such as the amount of service received by each core, it may not fully utilize the bandwidth due to the non-negligible synchronization overheads. Thus, we need to eliminate the synchronization overheads as much as possible.

Second to address application-level QoS requirements [5, 6], the I/O scheduler should provide core-level QoS first. That is, it needs to provide performance isolation between the cores so that each core can be guaranteed a fair bandwidth share. This core-level bandwidth partitioning is not a trivial problem for two reasons. First, it requires inter-core synchronizations. Second, the effective SSD bandwidth experienced by the system may greatly vary depending on the I/O workload types generated by the cores and the SSD internal activities such as garbage collection (GC) [7].

There have been a few efforts to cope with the above challenges [8, 9]. In [8], a new block layer for the Linux OS called block-mq has been proposed, which allows parallel dispatch of I/O requests between a filesystem and an NVMe-specific block driver. This framework demonstrates the scalable performance in the number of processor cores while leaving the scheduler implementation open to the developer's choice. In [9], an I/O scheduler called WABC, for use within NVMe SSDs, has been proposed. It is budget-based and uses a regression-based I/O cost estimation to provide QoS to each I/O queue. However, neither of the approaches directly addresses how to schedule I/O requests at the host level for the multi-queue SSDs. So far, for single-queue SSDs, several host-level I/O schedulers have been presented [5, 10-12]. For example, the FIOS [10] and FlashFQ [11] schedulers have been proposed to provide a fair bandwidth share to each I/O stream while the OIOS scheduler [5] has been proposed to satisfy multi-metric QoS requirements for virtual machines, including the fair bandwidth sharing. A similar fair-share scheduler called BCQ is also proposed in [12], but the multiple queues assumed by BCQ are for multiple channels within the SSD, not for multi-queue SSDs.

This paper proposes a novel host-level I/O scheduler called mqFlashFQ for the multi-queue SSDs. The proposed scheduler extends the existing FlashFQ [11] to the multi-queue structure, which is based on the depth-controlled start-time fair queuing called SFQ(D) [13]. In the mqFlashFQ, we use a randomization technique to decentralize the FlashFQ algorithm, thus achieving scalable performance in the number of processor cores while reducing the inter-core synchronization overheads. Moreover, we present an accurate calibration method to determine the cost of each I/O request in terms of its direction and size, which is necessary to provide performance isolation between the cores even with the varying SSD bandwidth. This calibration method is distinguishable from the previously proposed methods [9-12, 14] in that it makes it possible to provide a minimum bandwidth to each core while allowing any accidental extra bandwidth to be shared by all the cores assuming that the bandwidth demand by SSD internal activities such as GC can be estimated [15, 16]. Through our experiments with real NVMe SSD products, we demonstrate that the proposed mqFlashFQ and cost calibration method give a scalable performance and a fair bandwidth share to each core for various I/O workloads.

The rest of this paper's sections are as follows. In Section II, we summarize the NVMe specification and the FlashFQ. In Section III, we explain our proposed scheduler and calibration method in detail and present the experimental results in Section IV. In Section V, we describe the related work and in Section VI, the conclusion and future research directions.

II. PRELIMINARIES

A. Non-volatile Memory Express

The NVMe [4] is a recently proposed storage interface that allows the host OS to directly communicate with SSDs attached to the PCIe bus. This interface defines a set of standard commands and registers to be used between an NVMe driver of the host OS and the NVMe controller of an SSD. It features multiple I/O queues shared between the driver and the controller, as shown in Fig. 1, which is scalable in terms of the number of queues and the number of queue entries. The driver submits an I/O request to the controller through an I/O submission queue (SQ), and is notified of the request completion through an I/O completion queue (CQ) associated with the SQ. The interface supports up to 65,535 I/O SQs and CQs with up to 65,536 outstanding commands per I/O queue while it allows multiple SQs to share a CQ. To enable multiple cores to handle I/O queues in parallel, each I/O queue is

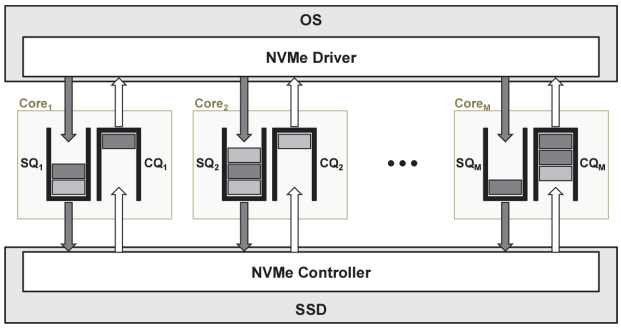


Fig. 1. The multi-queue structure of the NVMe inteface.

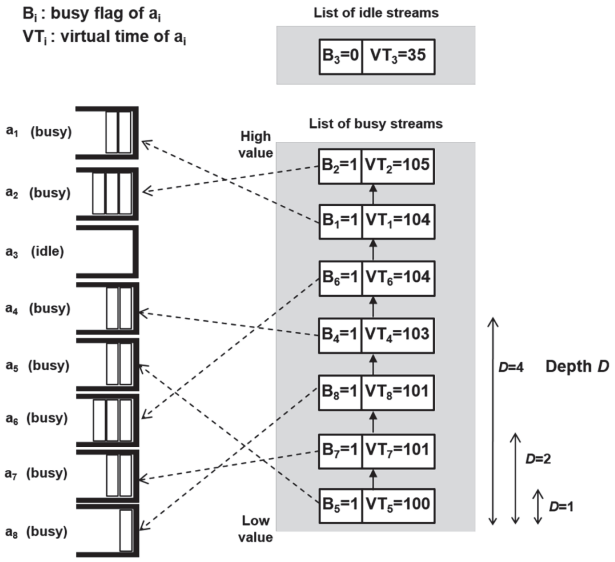


Fig. 2. An example of FlashFQ scheduling.

dedicated to a core possibly with a unique MSI/MSI-X interrupt assigned. Recently, to make use of NVMe SSDs, a next-generation block layer called block-mq [8] has been proposed for the Linux OS, which bridges between a filesystem and an NVMe-specific block driver. This new block layer replaces the existing single-queue structure with a multi-queue structure, which has become a bottleneck due to heavy lock contentions for the single queue by the multiple cores. It also proposes to use multiple software queues, each possibly associated with an application I/O stream, on top of multiple hardware queues, i.e., I/O submission queues, assuming that a set of software queues is scheduled on a dedicated hardware queue. This approach is promising as can be combined with various I/O schedulers to pursue application-level or system-level QoS metrics. For this approach to be successful, however, we should be able to provide a fair bandwidth share to each hardware queue, which is the scope of the paper.

B. FlashFQ Preliminaries

The FlashFQ is based on the depth-controlled start-time fair queueing SFQ(D) [13], which deals with single-queue storage with a certain degree of internal parallelism. That is, it allows up to D outstanding requests in service to better utilize the internal parallelism of the SSD while the original SFQ [17] allows one single request in service.

Apart from this tunable parameter D, the FlashFQ behaves the same as the SFQ. That is, to manage the progress of each I/O stream a_i in proportion to its assigned weight w_i, it maintains a virtual time VT_i for a_i, which is assumed to be initially set to a so-called system virtual time, explained later. Then, whenever a request $r_{i,j}$ from a_i is dispatched (or completed), it increases the VT_i by the weight normalized amount of the (storage) service time $s_{i,j}$ that is received by the request $r_{i,j}$, i.e., $VT_i = VT_i + s_{i,j}/w_i$. At this point, we define the start tag of $r_{i,j}$ as the value of VT_i before the increment and the finish tag to be the value after the increment. Then, when a new request $r_{i,j+1}$ arrives at a_i, the start tag of $r_{i,j+1}$ is determined depending on whether the previous request $r_{i,j}$ is in service or not at the arrival time of $r_{i,j+1}$. That is, if $r_{i,j}$ is in service, the start tag of $r_{i,j+1}$ is set equal to the finish tag of $r_{i,j}$. If $r_{i,j}$ is otherwise completed, meaning that a_i stays idle for a while, the virtual time VT_i for a_i is adjusted to the system virtual time $SVT(t)$, which is observed at the arrival time t of $r_{i,j+1}$. Therefore, the start tag of $r_{i,j+1}$ is set equal to the $SVT(t)$. The rationale behind this adjustment is that the SFQ recognizes a stream a_i that just becomes busy (or backlogged) after an idle period as a newly started one, thus treating the a_i to the same as busy streams by adjusting the VT_i to their VT_j's. However, the VT_j(t)'s of all the other busy streams observed at time t are not necessarily equal because not all the busy streams can be in service at every instant in real-world storage.

Therefore, the system virtual time $SVT(t)$ is usually defined as the minimum of the $VT_j(t)$'s of all the busy streams observed at t, i.e., $SVT(t) = \min\{VT_1(t), VT_2(t), ..., VT_N(t)\}$, and the newly backlogged stream a_i is treated as having the same progress as another busy stream with the minimum VT by setting $VT_i(t) = SVT(t)$. In this case, the start tag of $r_{i,j+1}$ in a_i, set to $SVT(t)$, is always greater than the finish tag of $r_{i,j}$ because a_i has made no progress for the idle period while other busy streams have.

With the above SFQ behavior in mind, we can now understand the FlashFQ. The FlashFQ concurrently dispatches a total of up to D requests from the busy streams. Fig. 2 shows a scheduling example of FlashFQ with depth D = 1, 2, 4. Each time one outstanding request is completed, another request can be dispatched to maintain D outstanding requests, which should be the first request in a stream with the minimum VT (or a request with the smallest start tag) among the busy streams. The main issue in the FlashFQ is how to compute the $SVT(t)$ with multiple outstanding requests in service, which will be used to determine the start tag of a new request arriving at

an idle stream. According to [11], one version of FlashFQ called Min-SFQ(D) takes the $SVT(t)$ as the minimum start tag of all the requests in service at time t while another version called Max-SFQ(D) takes their maximum start tag. As a side effect, the Min-SFQ (D) slowly advances $SVT(t)$ and may cause temporary unresponsiveness to other streams when a burst of requests arrives at a stream with a smaller virtual time. On the contrary, the Max-SFQ(D) quickly advances $SVT(t)$ and may cause a certain degree of unfairness between the newly backlogged and continuously backlogged streams.

One common limitation of the FlashFQ versions; however, is that they are a centralized decision algorithm where every dispatch requires all the values in the VT list of busy streams, shown in Fig. 2, not to be changed while determining a stream with the minimum VT (or a request with the minimum start tag). This is problematic in a multi-queue architecture because each core should acquire a global lock to access the VT's whenever it tries to dispatch a request, thus resulting in non-negligible inter-core synchronizations.

III. PROPOSED SCHEDULER

As seen from the related work, a flash I/O scheduler requires a scheduling mechanism and cost calibration method. In this section, we first describe the preliminaries of the FlashFQ and then detail the proposed mqFlashFQ and cost calibration method.

A. mqFlashFQ

The proposed mqFlashFQ extends the FlashFQ to the multi-queue architecture while eliminating the need for global locks. Since it is based on the SFQ, our scheduler inherits the good property [17] that it provides a fair bandwidth share even if the bandwidth varies as long as the cost for each request is accurate. This implies that if we can estimate the minimum bandwidth of the SSD, we may be able to guarantee a share of the minimum bandwidth to each core. For SSDs, the bandwidth variations result from two factors. One is the I/O workload types generated from the host system, which utilize different components within the SSD to different degrees. For example, read workloads typically bring much higher throughputs than write workloads because the induced flash operations of read requests require smaller resource usages than those of write requests. In the next subsection, we will investigate this aspect for various workload types in determining the cost of each I/O request. The other is SSD internal activities such as GC, which steal a certain portion of the SSD bandwidth intermittently. In this regard, we assume that an upper bound of the bandwidth used by GC can be identified for a finite time interval because within the SSD it is possible to estimate and

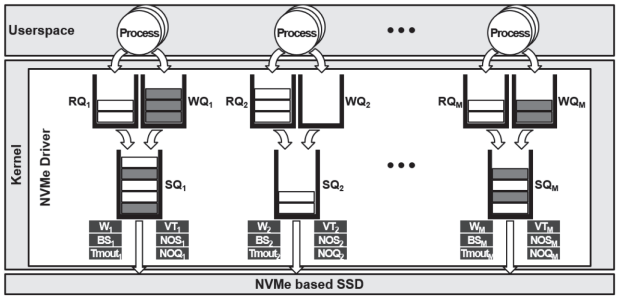


Fig. 3. Data structures used by mqFlashFQ.

inform the host OS of the bandwidth demand by the GC for that interval. This demand can be computed from the currently remaining free pages and the expected page consumption rate of the host workload during that interval [15, 16].

The main idea of the mqFlashFQ is to use a randomization technique so that each core can access the VT's of other cores in a lock-free manner, whenever needed. That is, assuming that each core i produces an aggregated I/O stream i, every backlogged core i randomly chooses any other backlogged core j and approximates $SVT(t)$ as the $VT_j(t)$ of core j at time t instead of searching for the minimum value in the VT list. This approach eliminates the need for a global lock to access the VT list, thus significantly reducing the inter-core synchronization overheads. Consequently, the proposed scheduler enables every core to make a scheduling decision in parallel, better exploiting the SSD bandwidth with an increasing number of cores. As a side effect, the above approach may compromise short-term fairness across the cores because some cores may repeatedly sample a value close to the maximum of the $VT_j(t)$'s, thus delaying a request dispatch. At the same time, other cores may repeatedly sample a value close to the minimum, thus encouraging an immediate dispatch. However, as shown in Section IV, despite such possibilities, long-term fairness across the cores will still be guaranteed because every core has equal chances of request dispatch in the probabilistic sense that it will sample the $SVT(t)$ under a uniform probability over the range between the minimum and maximum of VT_j's observed at the sampling moment t. Therefore, the proposed mqFlashFQ can achieve both scalable performance and performance isolation between the cores.

Fig. 3 shows the data structures used in our implementation of mqFlashFQ. In this figure, each core i has a dedicated pair of a submission queue SQ_i and a completion queue CQ_i and the CQ_i's are omitted for the sake of brevity. We assume that on top of a SQ_i there exist two additional software queues, RQ_i and WQ_i, where RQ_i is a queue for read requests and WQ_i is a queue for write requests generated from the core. For each of the three queues on core i, a set of variables is associated, i.e., w_i,

VT_i, BS_i, NOS_i, NOQ_i, and $Tmout_i$, although the set is not shown for RQ_i and WQ_i, for brevity. w_i is the weight assigned to a queue Q_i. VT_i is the virtual time defined in the previous subsection, representing the progress of Q_i. BS_i is the maximum batch size of a single dispatch in terms of the SSD bandwidth on Q_i. That is, each core i is allowed to dispatch as many requests $r_{i,k}$'s as possible at once as long as the total cost of the requests, i.e., $\sum_k s_{i,k}$, is less than or equal to BS_i where $s_{i,k}$ is the cost of the kth request $r_{i,k}$ to be dispatched in the batch by core i. This batch size is set proportional to the weight w_i, i.e., $w_i \times C$, where C is a tunable parameter called the base batch size. The value of C should be large enough to exploit the SSD bandwidth to the fullest extent. NOS_i is the total bytes of outstanding requests in service, dispatched from Q_i, which is defined as the difference between the total bytes of the requests dispatched and those completed. NOQ_i is the total bytes of queueing requests in Q_i. Finally, $Tmout_i$ is a timeout value used to determine the status of Q_i. That is, Q_i is determined to be idle if both NOS_i and NOQ_i remain zero for a longer period than $Tmout_i$. Otherwise, Q_i is determined to be backlogged. In our implementation, we use the same timeout value for all the queues, which is 1 ms. The features of our scheduler can be summarized as follows.

Random sampling for *SVT*(*t*): As explained above, whenever each core i needs to determine $SVT(t)$, the mqFlashFQ takes the $VT_j(t)$ of a randomly chosen core j (or SQ_j) among the backlogged cores. From each core's perspective, $SVT(t)$ needs to be determined in two cases. First, where a new request arrives at an idle queue, thus the start tag of the request should be determined from $SVT(t)$. Second, where a core i needs to determine whether it is eligible for dispatching a batch of requests from the SQ_i by checking if $VT_i(t) \leq SVT(t)$ In our scheduler, each core takes this test whenever (1) a request arrives at SQ_i, (2) a request dispatched from SQ_i is completed, or (3) a periodic timer associated with each core expires, which is responsible for periodic invocation of the scheduler with a period of 200 μs. In this test, each core references a randomly chosen busy core j in a lock-free manner, allowing the VT_j to be updated concurrently.

Type-wise cost accounting for each request: As indicated in [14], at the host level, it is difficult to accurately determine the SSD service time $s_{i,j}$ of a request $r_{i,j}$, only available within the SSD. This service time $s_{i,j}$ is required to calculate the VT_i of SQ_i (or the finish tag of the request $r_{i,j}$) for every request $r_{i,j}$ dispatched. Therefore, some host-level schedulers [10, 11] use the SSD response time of $r_{i,j}$ as an approximation of the service time, which is defined as the difference between its dispatch time and completion time. However, this makes the VT computation inaccurate because the SSD response time does not take into account parallel service times due to the parallel flash chips and may change depending on the I/O workload types generated by the host system. In the next subsection, we present an accurate calibration method to pre-estimate the cost of every request in terms of the storage resource usage, which can safely be used in the VT computation for any arbitrary workload. In our method, the cost of each request is estimated relative to that of a reference request, which is defined as a request of a certain direction and size under a reference access pattern.

Per-core queue separation between read and write requests: In the host OS, there are many cases where we may want to preferentially service read requests over write requests or vice versa. For example, synchronous read requests should be serviced with priority over asynchronous write requests because a synchronous read stream cannot generate any request without the completion of its previous request. Moreover, if we only use a single queue SQ_i for each core, there exist possibilities that read requests may be blocked by write requests in the queue, which is called read-blocked-by-write interference. To avoid this type of interference and preferentially service read or write requests, we use two software queues, i.e., RQ_i and WQ_i, on top of SQ_i and assign a weight value to each queue for another level of fair-share scheduling.

Per-queue handling of the deceptive idleness: In [18], the problem of deceptive idleness with storage devices has been addressed. That is, if an I/O scheduler hastily assumes that a stream has no further request when the stream is about to issue a request, and dispatches a request from another stream, the scheduler may violate the fair share guarantee. This is because whatever approximation used to determine $SVT(t)$ for real-world storage is essentially inaccurate compared to the ideal fluid server [17]. To alleviate this problem, our scheduler does not recognize as idle an idle period shorter than a threshold, thus not adjusting the $VT_i(t)$ of the associated SQ_i to $SVT(t)$ when a new request arrives. The threshold is a timeout value used for $Tmout_i$.

B. Accurate Cost Calibration

In this subsection, we describe our proposed cost calibration method, which is necessary for our scheduler to provide a fair-share guarantee to each core under various workload conditions. In the proposed method, the cost of every request is estimated in terms of storage resource usage, not in terms of the storage response time. This is based on an observation from our experiments with real multi-queue SSD products that the effective SSD bandwidth received by the cores can be factored into two: (1) a *guaranteed minimum bandwidth* determined under the condition that all possible sources of accidental extra bandwidth are identified, and (2) an *accidental extra bandwidth* obtained from opportunistic utilization of or opportunistic optimizations from SSD internal components. As can be seen in Section IV, the cost values obtained from our method still hold even when an arbitrary I/O workload other than the workload assumed

in the calibration is generated and/or mixed with a background workload generated within the SSD, e.g., GC.

To explain our observation, we give an analogy for multi-queue SSDs. In this analogy, we regard an SSD as a box and a request being serviced in the SSD as a ball in the box. A request with a different size and direction is treated as a ball with a different diameter. Then, the SSD bandwidth can be understood as the total space occupied by all the balls in the box. The amount of the occupied space is smaller than the box volume because there always exists a certain amount of empty space not occupied by the balls in the box. The occupied space can be increased or decreased depending on the combination of balls of different diameters. For example, it will shrink if we put only the largest balls into the box, and it will expand if we put only the smallest balls into the box. There may be certain combinations of different sizes of balls that maximally utilize the box volume, which we think may happen by chance.

In addition to the above analogy, in practice, we do not know either the box volume or the balls volume (or the occupied space by each ball). It is only possible for us to count how many balls of a size can be put into the box by taking a ball of a different size out of the box. With the above analogy and assumption in mind, let us think about how to fairly distribute the box volume to M users (or cores). For users to claim fair shares of the box volume, they need to accurately measure the real space occupied by the balls they put into the box. To this aim, given a ball of a reference size (or volume), the sizes of other balls are measured relative to the reference size. Note, however, that this measurement may result in different sizes for the same ball according to the ball combination in the box. If the box is full of large balls of the same size, and one is taken out of the box, the number of small balls of the same size that can be put into the box will be greater than expected with the actual occupied space by the large one because there exists a large amount of unoccupied space by the large balls. In this case, the size of the small ball would be underestimated. On the other hand, if the box is full of both large and small balls with the unoccupied space minimized, the number of small balls that can be traded for one large ball could be accurately measured. Thus, to accurately calibrate the cost of each request in a multi-queue SSD, we explore various combinations of requests being serviced in the SSD by synthetically generating requests of particular size and direction from each core.

Table 1 gives a cost calibration example with an Intel DC P3700 SSD [19] and a six-core processor (M = 6). Using Table 1, we want to determine the relative cost of a 16 kB read request (denoted by `RD16`) to that of a 16 kB write request (denoted by `WR16`), assuming the cost of `WR16` is one. To saturate the SSD bandwidth, we run a synthetic I/O benchmark called FIO [20] on each core consequently, continuously generating as many requests as possible in a random access pattern and the asynchronous I/O mode. In Table 1, for each configuration K, K cores generate `RD16` streams in a random access pattern (denoted by `rRD16`) while the remaining ($M – K$) cores generate `WR16` streams (denoted by `rWR16`).

Table 2 shows the resulting throughput received by each core and the relative cost of `RD16` determined for each of the seven configurations. In Table 2, for each configuration K, we denote the total throughput received by the K cores with `RD16` streams by `RD16`put(K) and that received by the ($M – K$) cores with `WR16` streams by `WR16`put (K). Thus, `WR16`put(0) gives the maximum total write throughput, i.e., 1065.1 MB/s, while `RD16`put(6) gives the maximum total read throughput, i.e., 2214.1 MB/s. From Table 2, we can see that configuration 5 gives the maximum relative cost of an `RD16` request to a `WR16` request, i.e., 0.6674, which we think is closest to the case where the unoccupied space in the box is minimized.

Fig. 4(a) shows the resulting throughput for each configuration K, received by each core after the maximum cost value is applied to the VT computation in our scheduler. From Table 2, we can see that every core receives exactly the $\frac{1}{6}$ of the total SSD bandwidth in terms of the cost-normalized throughput, which is obtained by summing up the cost of every request dispatched on each

Table 1. Workload configurations for Table 2's calibration

K	Core1	Core2	Core3	Core4	Core5	Core6
0	rWR16	rWR16	rWR16	rWR16	rWR16	rWR16
1	rWR16	rWR16	rWR16	rWR16	rWR16	rRD16
2	rWR16	rWR16	rWR16	rWR16	rRD16	rRD16
3	rWR16	rWR16	rWR16	rRD16	rRD16	rRD16
4	rWR16	rWR16	rRD16	rRD16	rRD16	rRD16
5	rWR16	rRD16	rRD16	rRD16	rRD16	rRD16
6	rRD16	rRD16	rRD16	rRD16	rRD16	rRD16

`r`=random access, `WR16`=16 kB write, `RD16`=16 kB read.

Table 2. A cost calibration example of RD16 relative to WR16 for Intel DC P3700 SSD (throughput: MB/s)

K	Core1	Core2	Core3	Core4	Core5	Core6	WR16put(*K*)	RD16put(*K*)	WR16traded(*K*)	rate(*K*)	RD16cost(*K*)
0	177.52	177.52	177.52	177.52	177.52	177.52	1065.10	0	0	-	-
1	197.45	197.45	197.45	197.45	197.45	197.45	987.25	197.45	77.85	253.63%	0.3943
2	215.68	215.68	215.68	215.68	215.68	215.68	862.73	431.37	202.37	213.16%	0.4691
3	229.17	229.17	229.17	229.17	229.17	229.17	687.50	687.50	377.60	182.07%	0.5492
4	240.28	240.28	240.28	240.28	240.28	240.28	480.57	961.13	584.53	164.43%	0.6082
5	245.60	245.60	245.60	245.60	245.60	245.58	245.60	1227.98	819.50	**149.85%**	**0.6674**
6	369.02	369.02	369.02	369.02	369.02	369.02	0	2214.10	1065.10	207.88%	0.4811

Bold font indicates that the value of RD16cost(K) obtained with the minimum rate(K) is chosen as the final cost.
WR16traded(*K*) = WR16put(*K*) – WR16put(*K*), rate(*K*) = RD16put(*K*)/WR16traded(*K*), RD16cost(*K*) =1/rate(*K*).

K	Core1	Core2	Core3	Core4	Core5	Core6	Total
Actual throughput (unit: MB/s)							
1	193.0	193.0	193.0	193.0	193.0	289.1	1253.9
2	196.6	196.6	196.6	196.6	294.5	294.5	1375.4
3	196.9	196.9	196.9	295.1	295.1	295.1	1475.9
4	197.7	197.7	296.3	296.3	296.3	296.3	1580.7
5	209.8	314.4	314.4	314.4	314.4	314.4	1781.7
6	389.8	389.8	389.8	389.8	389.8	389.6	2338.7
Cost-normalized throughput (unit: MB/s)							
1	193.0	193.0	193.0	193.0	193.0	193.0	1157.8
2	196.6	196.6	196.6	196.6	196.6	196.6	1179.5
3	196.9	196.9	196.9	196.9	196.9	196.9	1181.5
4	197.7	197.7	197.7	197.8	197.7	197.8	1186.5
5	209.8	209.8	209.8	209.8	209.8	209.8	1258.9
6	260.2	260.2	260.1	260.2	260.1	260.0	1560.8
Relative bandwidth (unit: %)							
1	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	100.00
2	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	100.00
3	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	100.00
4	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	100.00
5	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	100.00
6	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	**16.66**	100.00

(a)

K	Core1	Core2	Core3	Core4	Core5	Core6	Total
Actual throughput (unit: MB/s)							
1	186.1	186.1	186.1	186.1	186.1	352.3	1282.6
2	179.3	179.3	179.3	179.3	339.4	339.4	1395.9
3	173.6	173.5	173.6	328.6	328.6	328.6	1506.4
4	170.8	170.8	323.3	323.3	323.3	323.3	1634.8
5	177.0	335.1	335.1	335.1	335.1	335.1	1852.5
6	389.8	389.8	389.8	389.8	389.8	389.8	2338.7
Cost-normalized throughput (unit: MB/s)							
1	186.1	186.1	186.1	186.1	186.1	235.1	1165.4
2	179.3	179.3	179.3	179.3	226.5	226.5	1170.1
3	173.6	173.5	173.6	219.3	219.3	219.3	1178.5
4	170.8	170.8	215.8	215.8	215.8	215.8	1204.6
5	177.0	223.6	223.6	223.6	223.6	223.6	1295.2
6	260.1	260.1	260.1	260.1	260.1	260.1	1560.9
Relative bandwidth (unit: %)							
1	15.97	15.97	15.97	15.97	15.97	**20.17**	100.00
2	15.32	15.32	15.32	15.32	**19.36**	**19.36**	100.00
3	14.73	14.72	14.73	**18.61**	**18.61**	**18.61**	100.00
4	14.18	14.18	**17.91**	**17.91**	**17.91**	**17.91**	100.00
5	13.67	**17.27**	**17.27**	**17.27**	**17.27**	**17.27**	100.00
6	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	**16.67**	100.00

(b)

Fig. 4. Effect of the cost values on bandwidth partitioning under no GC. (a) The maximum cost value applied (RD16cost = 0.6674). (b) The average cost value applied (RD16cost = 0.5282).

core, multiplied by 16,384. On the contrary, from Fig. 4(b) where we use the average cost value, i.e., 0.5282, of the RD16 costs obtained for K = 1, 2, ..., 6, we can see that the minimum guarantee of the WR16 throughput of 177.52 MB/s is not possible. In the paper, we extend the above idea of using the maximum cost value of a request type to any arbitrary workload beyond the M + 1 workload configurations used to obtain the maximum values.

In our approach, to cover all the request types found in a real system, we used the proposed calibration method for each request type with a power-of-two size observed in the system, i.e., RD16, RD32, RD64, RD128, WR32, WR64 and WR128, assuming that WR16 is the common reference type. Below, we will explain how the reference type has been selected.

- A write request is selected as the reference type because it consumes more resources within SSDs than a read request of the same size due to the characteristics of flash memory [7], thus expected to be less sensitive to estimation errors.
- The I/O unit size used in the SSD, e.g., the size of a flash page, is selected as the reference size because requests smaller than the unit size are usually buffered or cached into a RAM buffer within the SSD to be serviced in a whole unit [7, 21]. Thus, for example, we assume that the cost of a WR4 (or RD4) request is simply set to $\frac{1}{4}\times$ the cost of a WR16 (or RD16) request calibrated above.
- Any request with a non-power-of-two size is assumed to be split into requests with power-of-two sizes by our scheduler.
- The access pattern is selected as random because in general random streams are known to give no better performance than sequential streams of the same type [7].

Finally, note that the minimum bandwidth to be guaranteed by the maximum cost values can be reduced by SSD internal activities such as GC stealing the bandwidth intermittently. However, in the next section, our experiments show that our scheduler still provides each core with a fair share of the reduced SSD bandwidth. Therefore, if we want to provide an absolute bandwidth guarantee, not the relative share guarantee, multi-queue SSDs should be designed in such a way that they can inform the host OS of the expected bandwidth consumption for each finite time interval [15, 16], so that the host OS

can compute the amount of bandwidth that can be guaranteed to each core during that interval.

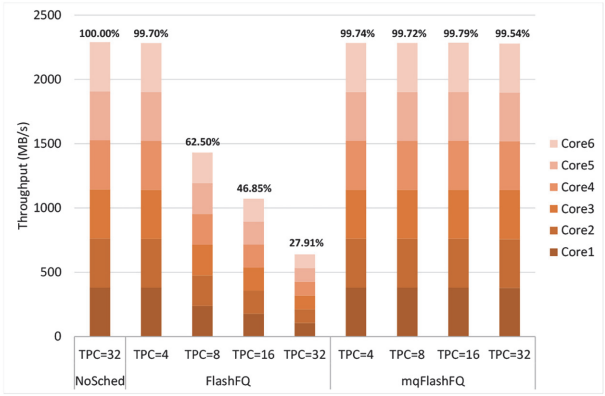
IV. EXPERIMENTAL RESULTS

A. Experimental Setup

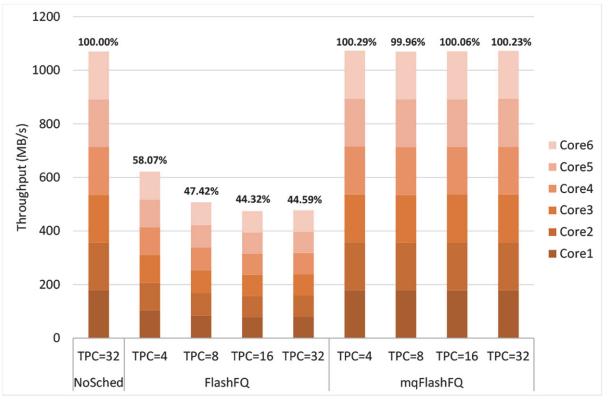
To evaluate the proposed mqFlashFQ and cost calibration method, we implemented them in the Linux 3.16.0 kernel using the plain open-source NVMe driver [22]. Our test bed is a deca-core computer system, which consists of an Intel Xeon processor E-2650v3 running at 2.3 GHz, 25 MB cache memory, 32 GB main memory and PCIe 3.0 × 16 lanes. All the target SSDs used in the experiments use four lanes, which are Intel DC P3700 of 400 GB [19], Intel 750 series of 400 GB [23] and Samsung 950 Pro of 256 GB [24], but we do not show the results of Intel 750 here, since there is not much difference between Intel DC P3700 and 750 in the performance behavior. To demonstrate the scalable performance and performance isolation between the cores, we use the synthetic I/O benchmark, i.e., FIO (Flexible I/O Tester) 2.2.10 with the libaio library, for the experiments shown in Sections IV-2 and IV-3. In this case, prior to each experiment, we perform a pre-conditioning on the target SSD so that every experiment can start from the same storage condition. Moreover, we configure the FIO jobs (= I/O threads) with a tunable FIO parameter, called the I/O depth, set to 64 per job, meaning the number of outstanding requests at the job level, and make them directly communicate with the NVMe driver bypassing the Linux ext4 filesystem. However, for the experiments shown in Section IV-4, we use real I/O workloads on the ext4 filesystem without any pre-conditioning. Finally, note that we ran the FIO workloads for 60 seconds for each experiment in Sections IV-2 and IV-3, which could avoid GC during the experiment on Intel DC P3700 but could not on Samsung 950 Pro, while we ran the real workloads for 20 minutes for the experiment in Section IV-4, which was accompanied by GC with Intel DC P3700.

B. Performance Scalability

In this subsection, we show that the proposed mqFlashFQ gives a scalable performance and outperforms a straightforward multi-queue extension of the FlashFQ using the shared *VT* list with a global lock. While the FlashFQ uses the simple cost calibration based on the storage response times, we use the cost values determined with our calibration. In this comparison, we vary the number of I/O threads per core, not the number of cores, to mimic an environment where a large number of cores contend for the global lock to access the *VT* list. In the experiment, we use six cores to generate I/O streams of the same type and a small base batch size of 49,152 for every SQ_i to



(a)



(b)

Fig. 5. Throughput comparison between FlashFQ and mqFlashFQ on Intel DC P3700 (TPC: Threads Per Core). (a) Total throughput obtained with `RD16` streams. (b) Total throughput obtained with `WR16` streams.

cause more scheduler invocations, which corresponds to a batch of 3 `WR16` requests or 4.5 `RD16` requests.

Fig. 5(a) shows the throughput received by each core generating `RD16` streams while varying the number of I/O threads per core (TPC) from 4 to 32. The FlashFQ shows a lower throughput with an increasing number of threads than the mqFlashFQ, while the latter shows almost no performance degradation in comparison with the case where no scheduler is applied, i.e., NoSched. We can observe a similar tendency from Fig. 5(b), which shows the throughput received by each core generating `WR16` streams with a varying number of TPC. From this, we can conclude that the proposed mqFlashFQ gives a scalable performance with little inter-core synchronization overheads.

C. Performance Isolation

In this subsection, we demonstrate that the proposed mqFlashFQ and cost calibration method provide performance isolation between the cores under various workload

Table 3. Composite workloads used in Section IV-3

Config	Core1	Core2	Core3	Core4	Core5	Core6
A	rWR16	rWR16	rRW16	rRW32	rRW64	rRW16
B	rWR16	rWR16	rRW16	rRW32	rRW64	rRW128
C	rWR16	rWR16	sRW16	sRW32	sRW64	sRW16
D	rWR16	rWR16	sRW16	sRW32	sRW64	sRW128
E	rWR16	rWR16	rRW16	rRW16	sRW64	sRW128
F	rWR16	rWR16	rRW16	sRW32	sRW64	sRW128

r=random access, s=sequential access, RD=read-only, WR=write-only, RW=read-write.

conditions on Intel DC P3700 and Samsung 950 Pro SSDs. In this experiment, we use six cores to generate composite workloads shown in Table 3, which include both random and sequential accesses for various request sizes. We used a large base batch size of 1,310,720 bytes for every SQ_i to saturate the allocated bandwidth to each core with a single I/O thread, which corresponds to a batch of 80 WR16 requests for both SSDs. For each core generating an RD or WR stream, we run only one I/O thread using FIO, but for each core generating an RW stream, we run two I/O threads, i.e., one for RD requests and the other for WR requests. Regarding the weight values of the cores, we apply two different settings to the scheduler. One is w_1:w_2:w_3:w_4:w_5:w_6 = 1:1:1:1:1:1 (symm-

Core	1	2	3	4	5	6	Total
Actual throughput (unit: MB/s)							
A	194.9	194.9	243.5	243.5	243.6	243.5	1363.9
B	192.9	192.9	241.0	241.1	241.2	241.3	1350.4
C	194.1	194.1	242.6	242.6	242.6	242.6	1358.6
D	195.1	195.1	243.8	244.0	243.8	244.1	1365.9
E	195.9	195.9	244.8	244.7	245.1	245.1	1371.4
F	195.4	195.4	244.2	244.2	244.3	244.4	1367.9
Cost-normalized throughput (unit: MB/s)							
A	194.9	194.9	194.9	194.9	195.0	194.9	1169.6
B	192.9	192.9	192.9	193.0	193.1	193.2	1158.0
C	194.1	194.1	194.2	194.2	194.2	194.2	1165.1
D	195.1	195.1	195.1	195.3	195.2	195.4	1171.2
E	195.9	195.9	195.9	195.9	196.2	196.2	1176.0
F	195.4	195.4	195.5	195.5	195.5	195.6	1173.0
Relative bandwidth (unit: %)							
A	16.67	16.66	16.66	16.67	16.68	16.66	100.00
B	16.66	16.66	16.66	16.67	16.68	16.69	100.00
C	16.67	16.66	16.66	16.67	16.67	16.66	100.00
D	16.66	16.66	16.66	16.67	16.67	16.68	100.00
E	16.66	16.66	16.66	16.66	16.68	16.69	100.00
F	16.66	16.66	16.66	16.67	16.67	16.68	100.00

(a)

Core	1	2	3	4	5	6	Total
Actual throughput (unit: MB/s)							
A	27.6	27.6	137.6	137.9	550.4	550.1	1431.2
B	28.0	28.0	139.8	139.8	559.2	559.4	1454.2
C	28.3	28.3	141.3	141.9	565.3	565.1	1470.1
D	29.8	29.8	145.3	145.3	577.2	577.3	1504.7
E	28.9	28.9	144.3	144.3	577.5	577.6	1501.6
F	29.5	29.5	145.6	145.6	580.1	579.9	1510.1
Cost-normalized throughput (unit: MB/s)							
A	27.6	27.6	110.1	110.4	440.6	440.4	1156.7
B	28.0	28.0	111.9	111.9	447.6	447.8	1175.2
C	28.3	28.3	113.1	113.5	452.5	452.4	1188.1
D	29.8	29.8	116.3	116.3	462.0	462.2	1216.4
E	28.9	28.9	115.5	115.5	462.3	462.4	1213.5
F	29.5	29.5	116.5	116.6	464.4	464.2	1220.6
Relative bandwidth (unit: %)							
A	2.38	2.38	9.52	9.54	38.09	38.07	100.00
B	2.38	2.38	9.52	9.52	38.09	38.10	100.00
C	2.38	2.38	9.52	9.55	38.09	38.07	100.00
D	2.45	2.45	9.56	9.56	37.98	38.00	100.00
E	2.38	2.38	9.52	9.52	38.09	38.10	100.00
F	2.42	2.42	9.54	9.55	38.04	38.03	100.00

(b)

Fig. 6. Bandwidth partitioning on Intel DC P3700 with no GC. (a) w_1:w_2:w_3:w_4:w_5:w_6 = 1:1:1:1:1:1. (b) w_1:w_2:w_3:w_4:w_5:w_6 = 1:1:4:4:16:16.

Core	1	2	3	4	5	6	Total
Actual throughput (unit: MB/s)							
A	8.0	8.0	45.1	48.2	49.1	45.1	203.5
B	8.3	8.3	46.7	50.1	51.0	58.1	222.6
C	8.6	8.6	48.0	51.0	52.2	48.0	216.3
D	8.9	8.9	49.7	53.1	54.0	61.3	236.0
E	8.7	8.6	48.6	48.5	52.8	60.0	227.2
F	9.2	9.2	51.9	55.4	56.3	64.0	246.0
Cost-normalized throughput (unit: MB/s)							
A	8.0	8.0	8.0	8.1	8.1	8.0	48.3
B	8.3	8.3	8.3	8.4	8.4	8.5	50.3
C	8.6	8.6	8.6	8.6	8.6	8.6	51.5
D	8.9	8.9	8.9	8.9	8.9	9.0	53.4
E	8.7	8.6	8.7	8.7	8.7	8.8	52.1
F	9.2	9.2	9.2	9.3	9.3	9.3	55.7
Relative bandwidth (unit: %)							
A	16.68	16.67	16.59	16.68	16.77	16.60	100.00
B	16.61	16.61	16.52	16.64	16.73	16.88	100.00
C	16.64	16.64	16.62	16.69	16.79	16.62	100.00
D	16.61	16.61	16.56	16.64	16.73	16.85	100.00
E	16.64	16.62	16.57	16.57	16.74	16.87	100.00
F	16.62	16.62	16.58	16.64	16.72	16.83	100.00

(a)

Core	1	2	3	4	5	6	Total
Actual throughput (unit: MB/s)							
A	1.4	1.4	30.0	32.2	130.5	118.9	314.5
B	1.3	1.3	29.4	31.1	127.8	144.7	335.8
C	1.4	1.4	30.3	32.4	130.7	119.5	315.7
D	1.4	1.4	31.6	33.7	136.9	155.9	361.0
E	1.4	1.4	31.0	31.0	131.1	147.4	343.4
F	1.4	1.4	31.4	33.6	136.2	155.1	359.2
Cost-normalized throughput (unit: MB/s)							
A	1.4	1.4	5.4	5.4	21.5	21.2	56.2
B	1.3	1.3	5.3	5.3	21.0	21.0	55.2
C	1.4	1.4	5.4	5.4	21.5	21.3	56.4
D	1.4	1.4	5.6	5.7	22.6	22.6	59.3
E	1.4	1.4	5.5	5.5	21.6	21.3	56.9
F	1.4	1.4	5.6	5.6	22.4	22.4	58.9
Relative bandwidth (unit: %)							
A	2.45	2.47	9.55	9.61	38.22	37.71	100.00
B	2.44	2.44	9.50	9.57	38.01	38.03	100.00
C	2.48	2.48	9.61	9.67	38.05	37.71	100.00
D	2.43	2.43	9.51	9.57	38.02	38.04	100.00
E	2.54	2.55	9.72	9.72	37.89	37.58	100.00
F	2.44	2.44	9.51	9.57	37.99	38.05	100.00

(b)

Fig. 7. Bandwidth partitioning on Samsung 950 Pro with GC. (a) w_1:w_2:w_3:w_4:w_5:w_6 = 1:1:1:1:1:1. (b) w_1:w_2:w_3:w_4:w_5:w_6 = 1:1:4:4:16:16.

etric weights), and the other is $w_1:w_2:w_3:w_4:w_5:w_6 = 1:1:4:4:16:16$ (asymmetric weights). For each workload configuration shown in Table 3, Fig. 6 shows the throughput and bandwidth received by each core on Intel DC P3700 SSD with no GC. As shown in Fig. 6(a), even for such a composite workload, each core receives exactly a $\frac{1}{6}$ of the total SSD bandwidth in terms of the cost-normalized throughput. as shown in Fig. 6(b), we can also observe a good bandwidth partitioning proportional to the asymmetric weights, noting that some extra bandwidth, observable in the different total cost-normalized throughputs ranging between 1156.7 and 1220.6 MB/s, is all shared by the cores in proportion to their weights.

On the other hand, Fig. 7 shows the throughput and bandwidth received by each core on Samsung 950 Pro SSD with GC for the same workload configurations and the same weight settings. On Samsung 950 Pro, we could not predict when the GC would be triggered even after we performed a pre-conditioning conditioning on the SSD. Despite the unpredictable GC, we can see from Fig. 7 that the proposed scheduler and cost calibration method still give a good bandwidth partitioning proportional to the weights although some trivial deviations from the expected partitioning are observed. In short, even if the total SSD bandwidth varies, the proposed scheduler and calibration method together give a fair bandwidth share to each core.

D. Evaluation with real I/O Workloads

To evaluate the real I/O workloads, we execute four VMware virtual machines (VMs), each running on a dedicated core. In each VM, we execute the Linux OS as the guest OS and ten kernel build jobs and eight FIO jobs as guest applications on top of the guest OS. We executed the FIO jobs in the VM because it is not possible to saturate the allocated bandwidth to each core with only the kernel build jobs because of the virtualization overheads. The FIO jobs were configured to generate different I/O types for different VMs, i.e., `rWR16` for VM1, `rRD32` for VM2, `rWR64` for VM3, and `rRD128` for VM4, with the FIO's I/O depth set to 128 for each job while the kernel build jobs were configured with their own build spaces. Regarding the scheduler parameters, we assigned

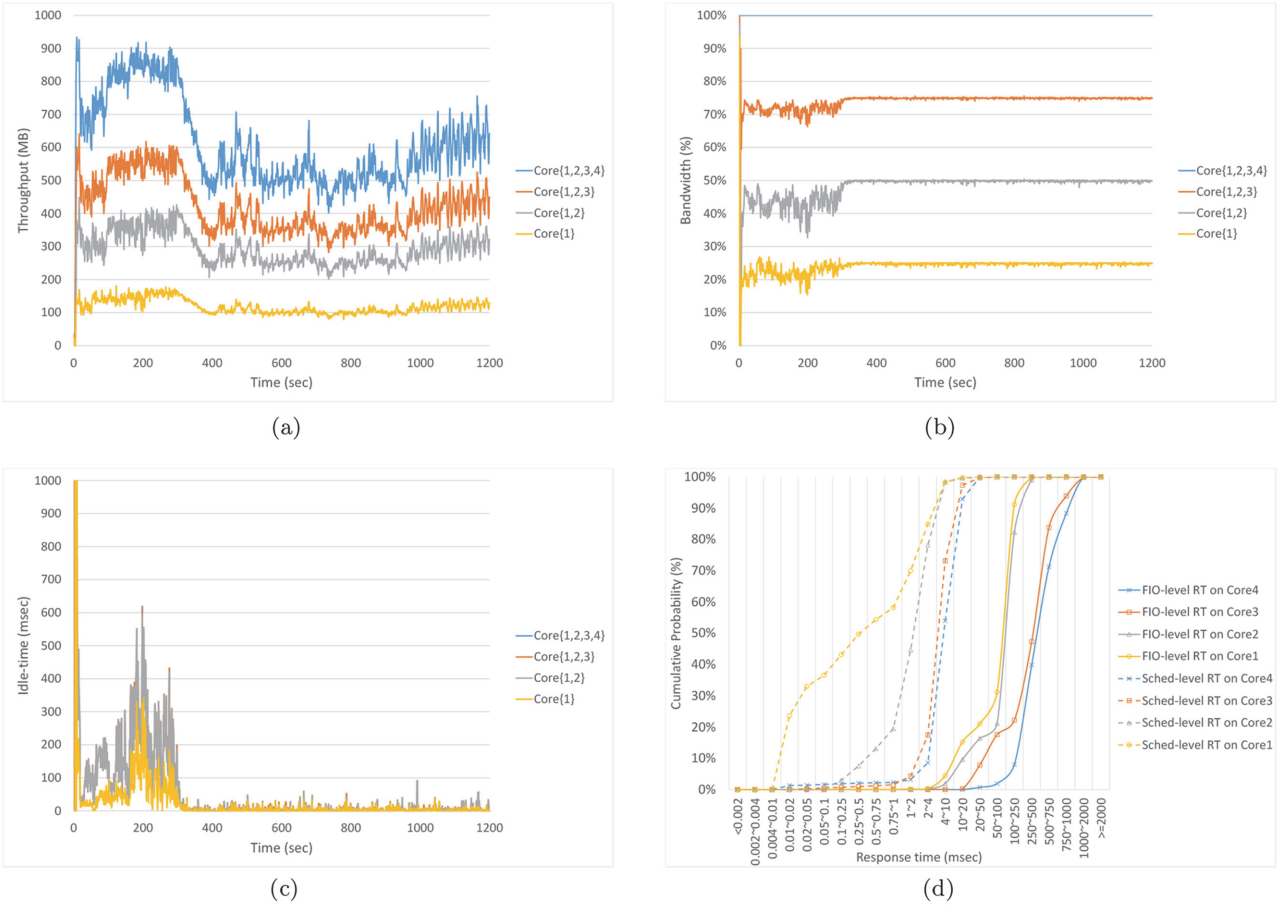


Fig. 8. Bandwidth partitioning between four virtual machines on Intel DC P3700. (a) Actual throughputs. (b) Relative bandwidths. (c) Idle times observed at SQ_i. (d) Response time distributions.

the same weight value to the four cores and used the same batch size used in the previous subsection. The guest Linux OS uses the ext4 filesystem and the default I/O scheduler, i.e., CFQ, which uses the single-queue structure for storage devices, while the host Linux OS uses the ext4 filesystem and our mqFlashFQ scheduler.

Fig. 8 shows the performance received by each core on Intel DC P3700 SSD, which was obtained for a 20-minute execution of the workload, accompanied by GC activities within the SSD. As shown in Fig. 8(a), we can see that the actual throughput received every second by each core i changes as time elapses, which is defined as the difference between the throughput labeled Core{1,…, i} and that of Core{1,…, i–1}. Note, however, that from Fig. 8(b) the relative bandwidth received by every core converges to 25% as the idle times caused by VM1 and VM2 almost disappear after about 300 seconds (see Fig. 8(c)).

These idle times seem to result from a mixture of the initial phase of the kernel build jobs and the small request sizes of the FIO jobs. Consequently, this prevents cores 1 and 2 from saturating the allocated bandwidth shares for the first 300 seconds but allows cores 3 and 4 to receive more bandwidth shares than 25%, as the SFQ originally intends. Finally, from Fig. 8(d), we can observe the response time distributions of the requests generated by each VM at the scheduler level and the FIO level, respectively. The scheduler-level response time of a request is the total time from arrival at SQ_i to completion, and the FIO-level response time is the total time from issue to notification of completion observed by the FIO. We can see that the scheduler-level response times are distributed between 10 μs and 50 ms while the FIO-level response times are distributed between 2 ms and 2,000 ms due to the virtualization overheads. Larger requests result in longer response times.

V. RELATED WORK

To provide quality service with flash storage, a lot of research on flash I/O scheduling has been done. The developed schedulers can be categorized into host-level schedulers [5, 10-12, 25] and device-level schedulers [9, 14, 26].

The host-level schedulers [5, 10-12, 25] schedule flash I/O requests within the host OS, thus have no control over their service times and the GC within the SSD. The FIOS (Fair Flash I/O Scheduler) [10] is a budget-based scheduler, which replenishes a predefined time budget to each I/O stream under certain conditions and consumes the budget by a pre-estimated cost whenever a request is dispatched. The FlashFQ (Fair Queueing I/O Scheduler for Flash-based SSDs) [11] is a fair queuing-based scheduler, which manages the progress of each I/O stream according to the SFQ(D) [13] by reflecting a pre-estimated cost for every request dispatch and adopts a throttled dispatch mechanism. According to [11], the FlashFQ gives better request response times than the FIOS because the former does not require the notion of a replenishment period as the latter does. Both schedulers use a calibration method in common to estimate the cost of each I/O request, which treats the target SSD as a black box. The method first estimates the storage response times of 4-KB request and a 128-kB request for each I/O direction, i.e., read or write, and takes the resulting values as their costs. In this case, the storage response time is defined as the total time from dispatch to completion assuming no interferences of other concurrent requests and GC. Then, it linearly interpolates the cost of an arbitrary-sized request between the estimated costs of a 4-kB request and a 128-kB request in the same direction as the considered request. However, this method is inaccurate because the storage response time does not take into account parallel service times due to parallel flash chips within the SSD. Therefore, the storage response time may change depending on I/O workload types generated by the host system [14], even if an alternative cost estimation is additionally presented in [10] that divides the storage response time by the number of outstanding requests in service in the SSD. Another example of host-level scheduling is the Budget allocation and Channel-based Queueing (BCQ) [12], which is budget-based and uses a regression-based cost estimation method. This method calculates the dynamic costs of a read request and a write request for every K profiling interval by solving K linear equations obtained, which may consider the dynamically changing SSD bandwidth. This scheduler, however, only considers such multi-channel SSDs, not multi-queue SSDs, that a direct mapping is possible with a simple hashing function between the address of a request and the flash channel within the target SSD that will service the request. As another example, the opportunistic I/O scheduler (OIOS) [5] is proposed for scheduling I/O streams from VMs, which uses weighted round-robin scheduling and dynamically adjustment of the time budget for each I/O stream using some feedback control. This scheduler, however, uses the same cost estimation as the FIOS. The vFair scheduler [25] is another host-level scheduler for VMs, which covers general storage devices including HDDs and SSDs. This scheduler consists of offline cost calibration and online feedback-based scheduling, which adjusts the offline predicted throughput for each virtual machine with a feedback control mechanism considering the run-time storage utilization and mix of different I/O types. The vFair's cost estimation method is similar to ours in that it models the I/O cost in terms of its direction, size and access pattern, but differs because the cost is defined on a VM basis while our cost is defined on a request basis.

The device-level schedulers schedule flash I/O requests within the SSDs, thus having the advantage of control over GC and accurate I/O cost estimation. One example

is the FACO scheduler proposed in the VSSD framework [14]. This scheduler is similar to the FIOS in that each I/O stream has a budget of credits to be replenished under certain conditions and to be consumed by a cost for each request. The main difference is that the FACO scheduler estimates the cost of each request as the sum of the actual service times of the subrequests composing the request, not the whole response time of the request, observed within the target SSD. For each write request, the cost may include that of the GC triggered by the request. Another example is the workload-aware budget compensation (WABC) scheduler [9], which is proposed for NVMe SSDs. This scheduler is also budget-based and uses the same regression-based cost estimation method as the BCQ. Both the FACO and WABC schedulers are distinguished from the host-level schedulers in that they may dedicate a group of flash blocks to each I/O stream and charge the GC overheads resulting from the group only to the associated I/O stream. The host interface I/O scheduler (HIOS) [26] is another example of device-level scheduling, which tries to provide uniform service times for as many requests as possible by redistributing the GC overheads over the requests with a positive slack time to their deadline. This approach, however, does not distinguish the I/O streams, thus providing no performance isolation between them.

All the above schedulers; however, do not consider host-level scheduling for multi-queue SSDs. The WABC scheduler deals with multi-queue SSDs, but at the device level. This study aims at the host-level scheduling on real multi-queue SSDs. Moreover, this paper presents a calibration method that accurately determines the cost of each request at the host level. Our goal is to provide a fair bandwidth share to each core while the overall performance received by the cores is scalable in the number of cores.

VI. CONCLUSION AND FUTURE WORK

This paper has proposed a fair-share flash I/O scheduler called mqFlashFQ for the multi-queue SSDs and an accurate cost calibration method to determine the cost of each request in terms of its direction and size. According to our evaluations with real NVMe SSD products, the proposed scheduler with the safe cost values determined by our calibration method guarantees a fair bandwidth share to each core despite the varying SSD bandwidth due to different workload types and garbage collection in the SSD. Moreover, it outperforms the FlashFQ in terms of the total throughput received by all the cores because the randomization technique significantly reduces the inter-core synchronization overheads. In our future work, we will consider integrating our scheduler into the block-mq layer of the Linux OS [8] and tuning it with more knowledge about the internals of the SSDs [27, 28].

ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (Grant No. NRF- 2021R1A2C2012837 and NRF-2021R1A4A1032252).

Conflict of Interest(COI)

The authors have declared that no competing interests exist.

REFERENCES

1. K. Guo, W. Zhao, Y, Lei, and Y. Gong, "I/O scheduling for solid state devices in virtual machines," in *Proceedings of 2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Indianapolis, IN, 2013, pp. 1-6.
2. S. Kim, D. Kang, and J. Choi, "Revisiting I/O scheduler for enhancing I/O fairness in virtualization systems," *IEICE TRANSACTIONS on Information and Systems*, vol. 97, no. 12, pp. 3133-3141, 2014.
3. Intel, "Serial ATA Advanced Host Controller Interface (AHCI) 1.3," 2008 [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/serial-ata-ahci-spec-rev1_3.pdf.
4. NVM Express Inc., "NVM Express revision 1.2 specification," 2014 [Online]. Available: https://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_2-Gold-20141209.pdf.
5. H. Park, S. Yoo, C. H. Hong, and C. Yoo, "Storage SLA guarantee with novel SSD I/O scheduler in virtualized data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2422-2434, 2016.
6. A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: enabling high-level SLOs on shared storage systems," in *Proceedings of the 3rd ACM Symposium on Cloud Computing*, San Jose, CA, 2012, pp. 1-14.
7. F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 181-192, 2009.
8. M. Bjorling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block IO: introducing multi-queue SSD access on multi-core systems," in *Proceedings of the 6th International Systems and Storage Conference*, Haifa, Israel, 2013, pp. 1-10.
9. B. Jun an D. Shin, "Workload-aware budget compensation scheduling for NVMe solid state drives," in *Proceedings of 2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, Hong Kong, 2015, pp. 1-6.
10. S. Park and K. Shen, "FIOS: a fair, efficient flash I/O scheduler," in *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, San Jose, CA, 2012.
11. K. Shen and S. Park, "FlashFQ: a fair queueing I/O scheduler for flash-based SSDs," in *Proceedings of 2013 USENIX Annual Technical Conference*, San Jose, CA, 2013, pp. 67-78.
12. Q. Zhang, D. Feng, F. Wang, and Y. Xie, "An efficient,

QoS-aware I/O scheduler for solid state drive," in *Proceedings of 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Zhangjiajie, China, 2013, pp. 1408-1415.
13. W. Jin, J. S. Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 37-48, 2004.
14. D. W. Chang, H. H. Chen, and W. J. Su, "VSSD: performance isolation in a solid-state drive," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 4, article no. 51, 2015. https://doi.org/10.1145/2755560
15. S. S. Hahn, J. Kim, and S. Lee, "To collect or not to collect: Just-in-time garbage collection for high-performance SSDs with long lifetimes," in *Proceedings of the 52Nd Annual Design Automation Conference*, San Francisco, CA, 2015, pp. 1-6.
16. L. P. Chang, T. W. Kuo, and S. W. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 4, pp. 837-863, 2004.
17. P. Goyal, H. M. Vin, and H. Cheng, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 690-704, 1997.
18. S. Seelam, R. Romero, P. Teller, and B. Buros, "Enhancements to Linux I/O scheduling," in *Proceedings of the Linux Symposium*, Ottawa, Canada, 2005, pp. 175-192.
19. Intel, "Intel Solid-State Drive DC P3700 series specification," 2015 [Online]. Available: https://tiscom.ru/sites/default/files/additional/ssd-dc-p3700-spec.pdf.
20. fio: Flexible IO Tester [Online]. Available: https://git.kernel.dk/cgit/fio/.
21. J. H. Yoon, E. H. Nam, Y. J. Seong, H. Kim, B. S. Kim, S. L. Min, and Y. Cho, "Chameleon: a high performance flash/FRAM hybrid solid state disk architecture," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 17-20, 2008.
22. NVM Express Drivers [Online]. Available: http://www.nvmexpress.org/drivers.
23. Intel, "Intel Solid-State Drive 750 series product specification," 2015 [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-750-spec.pdf.
24. Samsung, "Samsung SSD 950 Pro NVMe M.2 256 GB," 2023 [Online]. Available: http://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-950-pro-nvme-256gb-mz-v5p256bw.
25. H. Lu, B. Saltaformaggio, R. Kompella, and D. Xu, "vFair: latency-aware fair storage scheduling via per-IO cost-based differentiation," in *Proceedings of the 6th ACM Symposium on Cloud Computing*, Kohala Coast, HI, 2015, pp. 125-138.
26. M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, "HIOS: a host interface I/O scheduler for solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 289-300, 2014.
27. J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *Proceedings of the 7th ACM International Conference on Embedded Software*, Grenoble, France, 2009, pp. 295-304.
28. J. Kim, S. Seo, D. Jung, J. S. Kim, and J. Huh, "Parameter-aware I/O management for solid state disks (SSDs)," *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 636-649, 2012.

Hyeongseok Kang

Hyeongseok Kang received the B.S. and M.S. degrees in information communication from Soongsil University, Korea, in 2011 and 2013, respectively. He is currently a Ph.D. candidate in the Department of Information Communication, Soongsil University, Korea. His current research interests include real-time and embedded systems, flash storage systems, and motion control systems.

Kanghee Kim https://orcid.org/0000-0002-9321-9006

Kanghee Kim received the B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Korea, in 1996, 1998, and 2004, respectively. He is currently an associate professor in the School of Artificial Intelligence Software, Soongsil University, Korea. Previously, he was a senior engineer in the Mobile Communication Division, Samsung Electronics Co., Ltd. from 2004 to 2009. His current research interests include real-time and embedded systems, autonomous driving systems, and flash storage systems. Dr. Kim is a member of the IEEE.