Fair I/O Scheduler for Alleviating Read/Write Interference by Forced Unit Access in Flash Memory

Jieun Kim *
EE Dept., KAIST
Korea

Dohyun Kim *
EE Dept., KAIST
Korea

Youjip Won
EE Dept., KAIST
Korea

ABSTRACT

For the past few years, the enterprise Solid State Drives that employ NVM Express are widely used due to their high performance. It is common for multiple tenants and processes to share a single SSD. Providing fair SSD performance for multiple applications has become an important issue. We observe that the write request with `FUA` flag delays the processing of read request due to SSD internal read/write interference. To alleviate the performance degradation of the read requests, we propose `TABS`, per-**T**ype f**A**ir **B**andwidth I/O **S**cheduler for NVMe SSDs. `TABS` determines the fair bandwidth proportional to the maximum bandwidth and I/O issue rate for each type of request. (i) Two-phase Dynamic Scheduling sets fairness goals dynamically according to the I/O patterns of workload. Then, it throttles the FUA write requests to meet the pre-measured fairness goal. (ii) Software-based feedback makes more accurate scheduling possible. By using these techniques, `TABS` can guarantee fairness between the read and FUA flagged write requests. Finally, compared with the fairness goals, `TABS` achieves 76% fairness on average and up to 99.5% fairness, while the noop scheduler, the default Linux scheduler, shows 18% fairness.

CCS CONCEPTS

• **Information systems → Flash memory**.

KEYWORDS

I/O scheduler, I/O interference, NVMe SSD, flash memory, Forced Unit Access

*Both of the authors equally contributed to this work.

HotStorage '22, June 27–28, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539753

ACM Reference Format:
Jieun Kim, Dohyun Kim, and Youjip Won. 2022. Fair I/O Scheduler for Alleviating Read/Write Interference by Forced Unit Access in Flash Memory. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22), June 27–28, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3538643.3539753

1 INTRODUCTION

Modern Solid State Drives (SSD) can deliver more than 1 million operations per second. Multiple tenants often use a single SSD due to its high performance and low latency [12, 20, 22, 41]. SSDs can readily accommodate the I/Os from multiple applications with their internal hardware parallelism [8, 13, 27, 32, 42]. However, the interference among the applications that share the SSD still exists [9, 15, 22, 31].

Forced Unit Access (FUA) is a flag set for the I/O command. It was first mentioned in the SCSI specification [7]. When a device that supports the FUA receives a write command with the FUA flag, the data is directly reflected on the disk surface. The FUA allows the host to ensure data consistency without explicitly flushing the disk cache in the event of an unexpected power outage [3, 33]. The FUA is specified in the NVM express interface [6, 21].

There are the applications that use the FUA flagged write requests to ensure data durability [16, 17, 23, 28]. QEMU-KVM and Linux SQL Server provide the execution options for ensuring data durability with the FUA. `directsync` cache mode of QEMU-KVM uses both `O_DSYNC` and `O_DIRECT` semantics. The guest's I/O requests bypass the host page cache and are serviced with the FUA flag [1, 10, 19]. Microsoft SQL Server uses Write-Ahead Logging (WAL) to ensure ACID properties of transactions [37]. The write requests for the transaction's log records are dispatched to the disk with the FUA flag [4, 37]. When the transaction is committed, the log records are reflected directly on the disk.

The internal parallelism feature of the SSD flash chip allows it to process I/Os in parallel. However, in some cases, the I/Os are serialized (Sec. 2). If a flash read falls into the critical path of program execution, it is postponed due to the program, which has a 10-40 times longer latency than the read [5]. This is known as *read/write Interference* [34, 40].

We observe that when the host dispatches both FUA write (write request with FUA flag) and read requests, the read

latency becomes longer, similar to the FUA write (Sec. 3). It is unfair that the FUA writes result in excessively long latency for concurrent read requests. The read/write interference by the FUA writes is the cause of the unfairness. When the write request with FUA arrives at the device, a flash write is executed. When the program execution for the flash write causes the read/write interference, the read request is delayed. We call this phenomenon, *FUA interference*.

We propose TABS, per-***T***ype f***A***ir ***B***andwidth I/O ***S***cheduler to alleviate the performance unfairness caused by FUA interference. TABS provides the fairness between two types of I/O operations, read and FUA write, at the host level. TABS determines that the both requests are fair when their bandwidths are proportional to their maximum bandwidths and I/O issue rates. For example, when the read and FUA write requests are issued in a 1:1 ratio, it is fair if the read bandwidth becomes the half of its maximum bandwidth, as well as the FUA write. TABS separates the scheduling process into two phases. During the first phase, it identifies the I/O pattern and sets the fair bandwidth for each I/O type. During the second phase, it delays the dispatch of FUA write requests in order to adjust the number of outstanding FUA writes. TABS provides periodic feedback on the scheduling parameters for the accuracy. TABS keeps monitoring the actual bandwidth and evaluating whether the fairness is satisfying by the current scheduling policy.

TABS reduces the performance degradation of the read-intensive workloads even when other applications generate the FUA writes concurrently. It achieves 76% fairness on average and up to 99.5%, while the noop scheduler only reaches 18% fairness on average.

2 BACKGROUND & RELATED WORKS

2.1 SSD internal parallelism & read/write interference

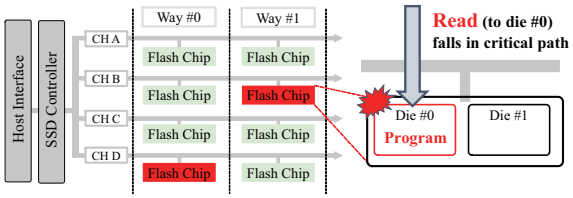


Figure 1: Read request delayed by P/E in the same die.

To maximize I/O parallelism, SSDs employ multiple flash chips in a multi-channel, multi-way structure [13, 42]. A channel is an independent I/O bus operated by a microprocessor. Each flash chip consists of one or more dies. A die consists of several planes. SSDs can process I/O operations in parallel by distributing them across the multiple channels, ways, dies, and planes. Flash memory performs read and write operations in the unit of flash page. For the flash write, the program of the corresponding pages is executed. Different types of flash commands, such as read or program, can be serialized in the same die [13]. This can degrade the SSD's internal hardware parallelism. Only the commands with the same type can be executed simultaneously in a single die. If the SSD controller tries to read some pages on the die where the program is executing, the read command cannot be executed until the program is completed [8, 27, 32]. A critical path is a flash chip component in which the different types of flash commands cannot be performed at the same time. SSD write latency is nearly 10-40 times that of read latency [5]. When the flash read falls into the critical path of write processing, it would be delayed (Fig. 1). This is referred to as SSD read/write interference [40].

2.2 Fair I/O Scheduling

Existing fair I/O schedulers are primarily designed to provide independent I/O performance across the multiple tasks that share the same storage device. FIOS [26] mentions how to resolve read/write interference at the host level. It adopts a naive approach, however, by dispatching the read request first and blocking all the write requests until the already dispatched read is complete. The write request can be continuously blocked due to the frequent read requests. Many studies have proposed the schedulers for providing the fair performance of the multiple tasks that share the same storage device [11, 25, 26, 30, 35, 36, 38]. These schedulers address the fairness among the processes that are running at the same time, but not with the fairness among the different types of I/O operations. There has been a study on scheduling the read/write fairness [18]. Lee et. al [18] isolates the read and write dispatch queues in the NVMe driver to reduce read/write interference. The ratio between the number of the read and write queues, which is statically set when the system boots, determines the performance of each type of I/O request. As a result, it is difficult to say that it is fair because the goal is always the same regardless of the I/O patterns. There have also been studies of the device-side I/O schedulers in which the device's controller schedules flash read and write operations [34, 39]. This approach, in comparison to scheduling at the host level, necessitates firmware modification, making it difficult for users to adopt.

3 MOTIVATION

3.1 FUA write interference

In this section, we observe and analyze the FUA interference with the read requests on OLTP benchmark in the virtualized environment. We use QEMU-KVM (6.2.91) [28] to create virtual machines. We use `sysbench` [14] as the OLTP benchmark and evaluate the MySQL [24] performance. A single

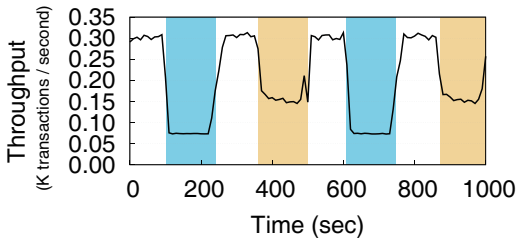


Figure 2: Throughput of OLTP benchmark on $VM_{readonly}$. $VM_{interference}$ performs OLTP write-only workload (Blue) and OLTP read-only workload (Yellow) alternately.

MySQL server runs on each virtual machine. When a virtual machine performs the `sysbench`, it executes the queries on its MySQL server. The virtual machines share the same NVMe storage device. Each virtual machine uses a logically independent 100GB partition as its virtual NVMe disk. The guest's MySQL server stores all the data in its virtual disk. We set the cache mode of the virtual NVMe disk to *directsync*. The FUA flag is set for all the write requests generated by the virtual machines. All requests bypass the host's page cache. We observe how the FUA write requests from the virtual machine affect the read performance of another virtual machine sharing the same SSD. The host and guest configurations are shown in Table 1.

		Guest		Host
		$VM_{readonly}$	$VM_{interference}$	
Software Configuration	OS	Centos 7.9 (Linux v5.8.5)		
	Applications	MySQL 8.0.29 Sysbench 1.1.0		QEMU-KVM 6.2.91
Hardware Configuration	CPU	1 vCPU	7 vCPUs	4 Cores (8threads)
	Memory	2GB	2GB	16GB
	Storage (Cache Mode)	Virtual NVMe Disk 100GB (directsync)		Samsung 970 Pro 512GB

Table 1: Configuration of the guests and host.

We generate one virtual machine which performs a read-intensive workload. We call it *$VM_{readonly}$*. $VM_{readonly}$ runs the OLTP read-only workload, which executes the read-only queries like SELECT. We create another virtual machine that interferes with $VM_{readonly}$. We call this virtual machine as *$VM_{interference}$*. To compare the interference caused by the read and the FUA write requests, $VM_{interference}$ alternately performs the FUA write-intensive workload and the read-intensive workload. $VM_{interference}$ uses the OLTP write-only for the FUA write-intensive workload and the OLTP read-only for the read-intensive workload. The OLTP write-only workload executes UPDATE, DELETE and INSERT queries. We set the 120 seconds gap between the two workloads. In this gap, $VM_{interference}$ does not operate any workloads. All the workloads access the database which has 1000 tables with 100,000 records each.

The performance of $VM_{readonly}$ varies greatly according to the I/O types generated by $VM_{interference}$. Fig. 2 shows the throughput of $VM_{readonly}$ when it concurrently operates with $VM_{interference}$. The blue and yellow regions represent the period that $VM_{interference}$ runs the FUA write-intensive and read-intensive workload, respectively. $VM_{interference}$ does not perform the workload in the white region; only $VM_{readonly}$ executes the OLTP read-only workload. When there is no workload running on $VM_{interference}$, $VM_{readonly}$ performs about 300 TPS. $VM_{readonly}$'s performance drops by around 43%, from 300 to 170 TPS, when $VM_{interference}$ performs the OLTP read-only workload simultaneously (Yellow region). In the blue area, the performance degradation of $VM_{readonly}$ is much worse. When $VM_{interference}$ performs the FUA write-intensive workload, $VM_{readonly}$'s performance drops by around 72 %, from 300 to 85 TPS. When virtual machines share the same SSD, the I/O interference caused by the FUA write requests is greater than the read requests.

3.2 Interference Analysis

User Type	I/O engine	# threads	I/O size	I/O depth	I/O type	Options
`Reader`	`libaio`	1	16 KB	128	random read	direct & sync
`Writer`	`libaio`	1	16 KB	128	random write	direct & sync

Table 2: Configuration for workers (fio).

To analyze the FUA interference in detail, we simulate an environment in which the FUA-intensive and read-intensive workloads are running simultaneously. To do this, we define two workers, `Reader` and `Writer`. Table 2 shows the configurations for each worker. Each worker runs as a single thread and uses the libaio engine. The libaio engine uses asynchronous I/O. We also set the direct option of `fio` to reduce the impact of the host's page cache. All I/O requests bypass the host page cache. Each worker issues an operation of size 16 KB. I/O depth is 128. `Reader` and `Writer` perform random read and write, respectively. We set the sync option of `fio` for the writers to be able to send the write with the FUA flag. When both direct and sync options are set to 1, the `O_DIRECT` and `O_DSYNC` flags are set on the open flag for the file. In Linux, if system call, `open()`, is invoked with `O_DIRECT` and `O_DSYNC`, the write request to file data is processed as direct I/O and the FUA flag is set [2].

We measure performance using 4 cores and 8 threads, 16 GB of DRAM machine, and an NVMe SSD (970 Pro) [29] as a storage device. We vary the number of workers and measure the bandwidth of read and write of each worker. For the convenience of explanation, the name of each workload is defined as $\{N_{writer}\}w\{N_{reader}\}r$ according to the number of workers. Then, we analyze the effect of FUA interference on the concurrent read. Fig. 3 shows the performance of a single reader in each situation where the reader runs with other workers; another reader, a writer (w-sync, w/ FUA), and a writer who does not set the sync option (w-nosync).

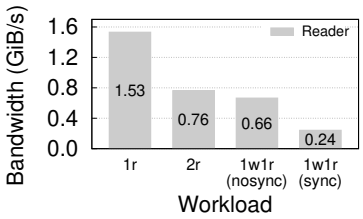


Figure 3: Read Bandwidth of a Reader with another worker.

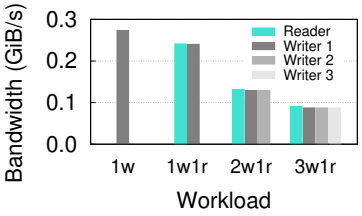


Figure 4: I/O Bandwidth of single Reader & multiple Writers.

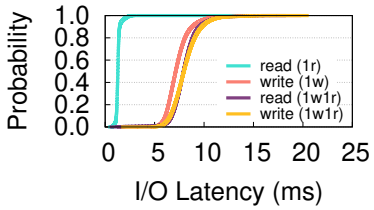


Figure 5: I/O Latency for 1r, 1w and 1w1r.

For 1w1r-nosync, the write request is the direct I/O and the FUA flag is not set. The write requests are only transferred to the disk's writeback cache. We'll call it `cache writer`.

The drop in read performance caused by FUA writes is noticeable. When the reader is added to a single reader (2r), the single reader's bandwidth is reduced to 0.76 GiB/s, which is 50% of the original capacity. Two readers occupy the bandwidth equally. When the cache writer is added, it is reduced by about 57% to 0.66 GiB/s. When the writer is running concurrently, the bandwidth decrease of a single reader is substantially more than when running with the reader or cache writer.

We find that due to the FUA interference, the reader's performance falls to a level comparable to the FUA write. Fig. 4 shows the bandwidth of each worker when the number of concurrently operating readers is fixed to one and the number of writers is changed. For comparison, the result of 1w is added. Surprisingly, regardless of the operation types, all workers have comparable bandwidth. Each worker seems to have fair device utilization, but they do not. In the case of 1w1r, the reader's bandwidth is reduced by 84.3%, while the writer only decreases by 12% from 0.27 GiB/s to 0.24 GiB/s. The latency of the read request, which is quite low in comparison to the write, becomes nearly the same as the write (Fig. 5). The reader's bandwidth gets worse as the number of writers increases (Fig. 4). For workload 3w1r, the reader's bandwidth is only about 5.8% of the maximum read bandwidth.

3.3 Outstanding FUA writes

We measure how the interference behavior changes when we reduce the number of outstanding FUA writes. We change the I/O depth of the writer and observe how the bandwidth of each worker changes. I/O depth is the maximum number of requests a worker can issue asynchronously. As it is reduced, the number of outstanding I/O generated by the worker decreases as well. The bandwidth of concurrent read operations also increases due to the alleviation of FUA write interference. Fig. 6 shows the read and write bandwidth when the writer's I/O depth varies. In 1w1r and 3w1r, when the I/O depth becomes less than 64 and 20, respectively, the read bandwidth rises sharply. FUA interference is alleviated by reducing the number of outstanding FUA writes. The desired read performance can be achieved by adjusting the number of outstanding FUA writes.

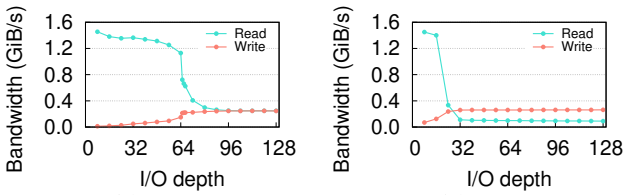


Figure 6: I/O Bandwidth of each type of I/O with varying writers' I/O depth.

4 DESIGN & IMPLEMENTATION

4.1 Overview & Goal

We find that reducing the number of outstanding FUAs at the host level can alleviate read/write interference in the SSD. We propose newly developed I/O scheduler for flash memory, particularly NVMe SSDs. `TABS` delays the dispatch of FUA write to obtain the proper bandwidth for both read and FUA write requests. `TABS`, per-type fair bandwidth I/O scheduler, guarantees that read and FUA write requests use the device fairly. `TABS` gives the different bandwidth weights to read and FUA write requests. Each of their bandwidth must be proportional to their respective weights to achieve fairness. The weights of read or FUA write are set to be proportional to their maximum bandwidth (BW_{Max}) and I/O issue rate ($\mathcal{I}$). Accordingly, the fair bandwidth (FBW) is defined as Eq. 1.

$$FBW_x = BW_{Max,x} \times \frac{\mathcal{I}_x}{\mathcal{I}_{total}} \quad (1)$$

In `TABS`, we use two techniques, (i) Two-Phase Dynamic Scheduling and (ii) Software-based Feedback. Through these two techniques, `TABS` achieves the fair bandwidth dynamically with low overhead from the I/O scheduler. Fig. 7 shows the overview of `TABS`.

4.2 Two-Phase Dynamic Scheduling

`TABS` divides the scheduling process into two phases; IDLE and SCHED. During the IDLE phase, it observes the current I/O pattern in order to reflect the fairness goals immediately based on the workload characteristics. The collected

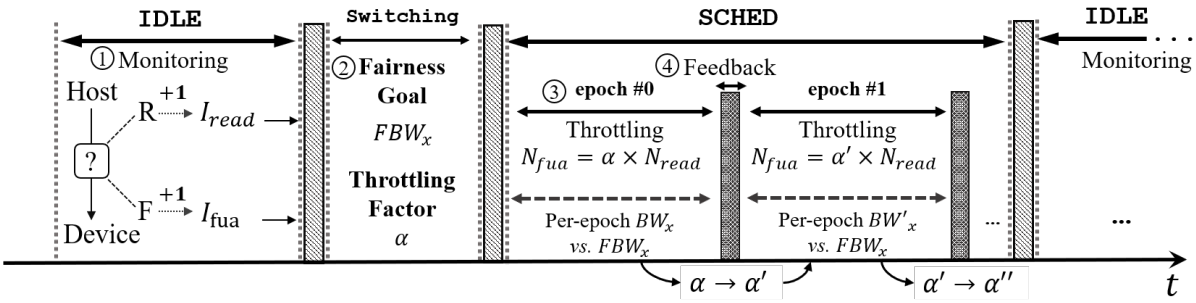


Figure 7: TABS Overview.

information on I/O pattern may differ from the actual one if the scheduler is working. To avoid this problem, the I/O scheduling is disabled during this period. TABS throttles the dispatch of the FUA write requests during the SCHED phase. We define the *throttling factor (α)*. The number of dispatched FUA write (N_{fua}) is limited to the number of dispatched read requests (N_{read}) multiplied by the *throttling factor*.

IDLE Phase During this phase, TABS recognizes the current workload's I/O pattern. It counts the number of issued I/O requests [1] from the host (① in Fig. 7). When enough I/O statistics have been collected, scheduling can begin. When the number of issued I/Os collected by TABS exceeds the threshold, the phase is switched from IDLE to SCHED (② in Fig. 7). The statistics collected by TABS are periodically reset. The reset period and threshold should be configured according to their system environments, such as the spec of the storage device. When the phase is switched, the scheduler delivers the fair bandwidth (Eq. 1) and throttling factor. Both are calculated based on the number of issued I/O requests. The throttling factor is initialized by dividing the number of read requests by the number of write requests.

SCHED Phase During this phase, the issued I/O requests from the host are scheduled (③ in Fig. 7). The requests inserted into the NVMe device driver's submission queue (SQ) are dispatched to storage in the order in which they are queued. TABS does not schedule any read requests. On the other hand, the FUA write requests are scheduled to meet the fairness goal. TABS inserts the FUA write requests into a newly created queue, the *FUA delay queue*. Only a certain number of queued FUA write requests are moved to SQ and given a chance to dispatch.

SCHED phase is composed of multiple epochs. The epoch is a time period. In the SCHED phase, the scheduler measures the actual bandwidth in the unit of epoch. Then, it processes the feedback to readjust the throttling factor by comparing the per-epoch bandwidth to the fair bandwidth (④ in Fig. 7). Through the feedback process, the scheduler keeps regulating the throttling factor that can reach the fair bandwidth.

[1] the number of I/O requests that inserted into SQ of NVMe device drive.

4.3 Software-based Feedback

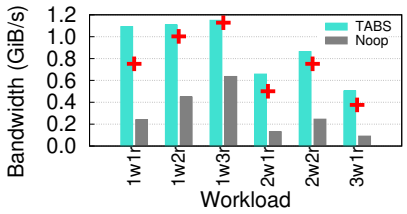
In this section, we explain the detail of the feedback process. TABS compares the per-epoch bandwidth to the fair bandwidth every end of the epoch. Once the per-epoch bandwidth is less than 90% of the fair bandwidth. it re-calculates the fairness factor. To do this, we first calculate the change ratio of the throttling factor. The factor must change at a rate of the change ratio. The change ratio is calculated by the per-epoch bandwidth divided by the fair bandwidth. It is always lower than one. The scheduler calculates and applies the change ratio for each I/O type, one by one. For read requests, TABS adjusts the throttling factor by multiplying the change ratio and original factor to make it smaller. On the other hand, for FUA write requests, TABS divides the original factor by the change ratio.

5 EVALUATION

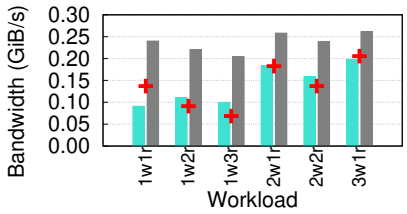
5.1 Experiment Setup

Here, we examine the bandwidth and fairness achievement of TABS on a machine with 4 cores and 8 threads (Intel(R) Core(TM) i7-4790 CPU with 3.60GHz) and 16 GByte DRAM. The CentOS 7.4 (kernel v5.8.5) and Samsung 970 Pro 512GB [29] are used. To measure the bandwidth of each type of I/O request, we define the two types of workers, the same as those used in Section 3. All workers bypass the host page cache. `Writer` sends the random write with FUA flags and `Reader` sends the random read (Table 2). The number of active cores is set to be the same as the number of workers for all workloads.

We evaluate TABS and noop scheduler, the default scheduler for Linux kernel. In order to verify the fairness, we compare the bandwidth results with the calculated fair bandwidth (Eq. 1) for all workloads. For the maximum bandwidth of each I/O type, we use the pre-measured value; 1530 MiB/s for read and 273 MiB/s for FUA write. The I/O issue rate ratio is assumed to be the same as the ratio between the number of `Reader` and `Writer`. Lastly, the static values for TABS are as follows: During the IDLE Phase, the reset period is 500 ms and the threshold is 5,000. Once the total count is over 5,000 in 500 ms, TABS switches the phase from IDLE to SCHED.



(a) Total bandwidth of read



(b) Total bandwidth of write

Figure 8: I/O Bandwidth of TABS & Noop, ✚: Fair Bandwidth of workload.

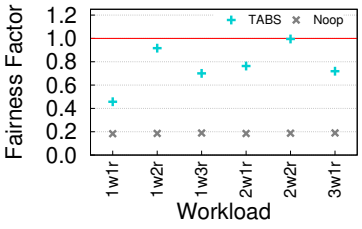


Figure 9: Fairness Factor (Eq. 2)

The SCHED Phase lasts 50 seconds and each epoch is set to be 1 second. TABS regulates the throttling factor every 1 second.

5.2 I/O Scheduling

To confirm that the fairness is achieved through TABS, we measure the bandwidths of the read and FUA write requests with varying the number of two workers. We omit the bandwidth of each worker since TABS only considers the fair bandwidth of each I/O type. The bandwidth results for all workloads we evaluated are shown in Fig. 8. Fig. 8a and Fig. 8b illustrate the bandwidths of the read and FUA write requests, respectively. Due to the lack of active cores, we only analyze workload for up to four workers. The fair bandwidths of the workloads are also shown in Fig. 8 for comparison. As we previously stated, the fair bandwidth is based on the reader and writer ratio.

In all workloads except *1w1r*, the bandwidths of both types of I/O requests in TABS are similar to or higher than their own fair bandwidth (Fig. 8). TABS achieves the fairness goal that we define. IDLE phase detects the current I/O pattern of executed workload successfully without any pre-setting for the workload. TABS can achieve the proper fairness for all the workloads, because its goal changes dynamically according to the current I/O pattern. The fair I/O scheduler, TABS, provides more opportunities to obtain higher performance for the I/O type that is generally faster and quicker. It gives the device the possibility to accommodate more I/O traffic even if the FUA requests are issued frequently.

We calculate the error rate of each I/O type and workload. The error rate is the normalized error to fair bandwidth, which is calculated from the absolute difference between actual and fair bandwidth. The average error rates for the other workloads except *1w1r* are 0.18 and 0.15 for read and FUA write, respectively. The error rates of workload *1w1r* are 0.40 and 0.34. This is more than double that of the other workloads. However, even though it has the largest errors, it is still substantially fairer than the noop scheduler (Fig. 9).

5.3 Fairness Factor

To compare the fairness between the noop scheduler and TABS, we define the *fairness factor* ($\mathcal{F}$) (Eq. 2). The fairness factor represents how fair the two types of I/O requests are in terms of bandwidth. The read/FUA write operations become fairer as the fairness factor approaches one. When the factor is one, we assume that the performance of all types of I/Os is 100% fair.

$$\mathcal{F} = \frac{BW_{read}/\{BW_{Max,read} \times I_{read}\}}{BW_{fua}/\{BW_{Max,fua} \times I_{fua}\}} \quad (2)$$

The fairness factors of the noop scheduler and TABS for all workloads are shown in Fig. 9. For comparing the result easier, we inverse the value when it is more than one. Factor calculation is done with Eq. 2, but if the factor is 1 or more, the reciprocal number of the result is taken for ease of verification. The average fairness factor is 76% for TABS and 18% for noop scheduler. TABS shows the 4× higher fairness than the noop scheduler.

6 CONCLUSION

In modern NVMe SSD, multiple outstanding FUAs degrade concurrent read's performance. When the FUA write arrives at the device, it occurs the flash write inside the SSD. The flash read/write interference induced by the FUA is the cause of the unfairness. We propose a fair I/O scheduler for NVMe SSDs, TABS, that delays the FUA write dispatch to obtain the proper bandwidth for both read and FUA write requests. TABS provides fairness between different types of I/Os at the host level. To ensure the fairness according to the request type, the bandwidth of each request is set to be proportional to its maximum bandwidth. The scheduler process is composed of two phases; IDLE and SCHED. TABS serves the different fairness goals according to the workloads. Lastly, it achieves 4 × higher fairness than the noop scheduler.

7 ACKNOWLEDGEMENTS

We would like to thank Bryan S. Kim, our shepherd, for his assistance in finalizing our paper. We would also like to thank the anonymous reviewers for their insightful comments, which made us improve the paper. This work was supported by IITP, Korea (No. 2018-0-00549), NRF, Korea (No. NRF-2020R1A2C3008525), and SNU-SK Hynix Solution Research Center (S3RC) (No. MOUS002S).

REFERENCES

[1] Dulcardo Arteaga and Ming Zhao. 2014. Client-side flash caching for cloud systems. In *Proc. of ACM SYSTOR*. 1–11.

[2] Dave Chinner. 2018. iomap: Use FUA for pure data O_DSYNC DIO writes. https://patchwork.kernel.org/project/linux-fsdevel/patch/20180418040828.18165-5-david@fromorbit.com/.

[3] Jonathan Corbet. 2010. The end of block barriers. https://lwn.net/Articles/400541/.

[4] Robert Dorr. 2022. SQL Server On Linux: Forced Unit Access (Fua) Internals. https://techcommunity.microsoft.com/t5/sql-server-blog/sql-server-on-linux-forced-unit-access-fua-internals/ba-p/3199102.

[5] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T Kandemir, Chita R Das, and Myoungsoo Jung. 2017. Exploiting intra-request slack to improve SSD performance. In *Proc. of ASPLOS*. 375–388.

[6] NVM Express. 2019. NVM Express Base Specification Revision 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[7] Fibre Channel FC and Serial Attached SCSI SAS. 2016. SCSI Commands Reference Manual. (2016).

[8] Congming Gao, Liang Shi, Kai Liu, Chun Jason Xue, Jun Yang, and Youtao Zhang. 2020. Boosting the performance of SSDs via fully exploiting the plane level parallelism. *IEEE TPDS* 31, 9 (2020), 2185–2200.

[9] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. 2006. Enforcing performance isolation across virtual machines in xen. In *Proc. of ACM/IFIP/USENIX Middleware*. Springer, 342–362.

[10] Son-Hai Ha, Daniele Venzano, Patrick Brown, and Pietro Michiardi. 2016. On the impact of virtualization on the I/O performance of analytic workloads. In *Proc. of CloudTech*. IEEE, 31–38.

[11] Wei Jin, Jeffrey S Chase, and Jasleen Kaur. 2004. Interposed proportional sharing for a storage service utility. *ACM SIGMETRICS PER* 32, 1 (2004), 37–48.

[12] Byunghei Jun and Dongkun Shin. 2015. Workload-aware budget compensation scheduling for NVMe solid state drives. In *Proc. of NVMSA*. IEEE, 1–6.

[13] Myoungsoo Jung and Mahmut T Kandemir. 2012. An Evaluation of Different Page Allocation Strategies on High-Speed SSDs. In *Proc. of USENIX HotStorage*.

[14] Alexey Kopytov. 2004. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/* (2004).

[15] Norbert Kuck, Harald Kuck, Edgar Lott, Christoph Rohland, and Oliver Schmidt. 2002. SAP VM Container: Using process attachable virtual machines to provide isolation and scalability for large servers. *Article, SAP AG, Walldorf, Germany* 2 (2002).

[16] KVM. 2017. KVM Disk Cache Modes. (2017). https://doc.opensuse.org/documentation/leap/virtualization/html/book-virtualization/cha-cachemodes.html.

[17] KVM. 2021. KVM, Kernel-based Virtual Machine. https://www.linux-kvm.org/page/Main_Page.

[18] Minkyeong Lee, Dong Hyun Kang, Minho Lee, and Young Ik Eom. 2017. Improving read performance by isolating multiple queues in NVMe SSDs. In *Proc. of IMCOM*. 1–6.

[19] Xiaofei Liao, Hai Jin, Xuhong Wang, Bingbing Zhou, and Dingding Li. 2013. VDB: Virtualizing the On-Board Disk Write Cache. In *Proc. of HPCC & EUC*. IEEE, 948–955.

[20] Renping Liu, Xianzhang Chen, Yujuan Tan, Runyu Zhang, Liang Liang, and Duo Liu. 2020. SSDKeeper: Self-adapting channel allocation to improve the performance of SSD devices. In *Proc. of IPDPS*. IEEE, 966–975.

[21] Shane Matthews. 2015. NVM Express: SCSI Translation Reference. *NVM Express Workgroup* (2015), 1–54.

[22] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Julie Lee, Lukas Rupprecht, Dimitris Skourtis, Yang Yang, and Erez Zadok. 2021. CNSBench: A Cloud Native Storage Benchmark. In *Proc. of USENIX FAST*. 263–276.

[23] Microsoft. 2022. Microsoft SQL Server. https://www.microsoft.com/ko-kr/sql-server/sql-server-2022.

[24] MySQL. 2022. MySQL Documentation. https://dev.mysql.com/doc/.

[25] Chandandeep Singh Pabla. 2009. Completely fair scheduler. *Linux Journal* 184 (2009), 4.

[26] Stan Park and Kai Shen. 2012. FIOS: a fair, efficient flash I/O scheduler.. In *Proc. of USENIX FAST*, Vol. 12. 13–13.

[27] Seon-yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. 2010. Exploiting internal parallelism of flash-based SSDs. *IEEE Computer Architecture Letters* 9, 1 (2010), 9–12.

[28] QEMU. 2022. QEMU, Open Source Processor Emulation. http://www.qemu.org.

[29] Samsung. 2018. Samsung V-NAND SSD 970 PRO. https://s3.ap-northeast-2.amazonaws.com/global.semi.static/Samsung_NVMe_SSD_970_PRO_Data_Sheet_Rev.1.0.pdf.

[30] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proc. of USENIX ATC*. 67–78.

[31] Gaurav Somani and Sanjay Chaudhary. 2009. Application performance isolation in virtualization. In *Proc. of CLOUD*. IEEE, 41–48.

[32] Xiang Song, Jian Yang, and Haibo Chen. 2013. Architecting flash-based solid-state drive for high-performance i/o virtualization. *IEEE Computer Architecture Letters* 13, 2 (2013), 61–64.

[33] Curtis E. Stevens. 2008. Information technology - AT Attachment 8 - ATA/ATAPI Command Set (ATA8-ACS). https://web.archive.org/web/20200806025823/http://www.t13.org/Documents/UploadedDocuments/docs2008/D1699r6a-ATA8-ACS.pdf.

[34] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling fairness and enhancing performance in modern NVMe solid state drives. In *proc. of ACM ISCA*. IEEE, 397–410.

[35] Paolo Valente. 2019. Budget Fair Queueing. https://lwn.net/Articles/784267/.

[36] Carl A Waldspurger and William E Weihl. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of USENIX OSDI*. 1–es.

[37] Dennis White. 2013. Microsoft® SQL Server® Always on I/O Reliability Storage System on Hitachi Virtual Storage Platform. (2013).

[38] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. 2021. D2FQ:Device-Direct Fair Queueing for NVMe SSDs. In *Proc. of USENIX FAST*. 403–415.

[39] Guanying Wu and Xubin He. 2012. Reducing SSD read latency via NAND flash program and erase suspension.. In *Proc. of USENIX FAST*, Vol. 12. 10–10.

[40] Suzhen Wu, Weiwei Zhang, Bo Mao, and Hong Jiang. 2019. Hotr: Alleviating read/write interference with hot read data replication for flash storage. In *Proc. of DATE*. IEEE, 1367–1372.

[41] Minhoon Yi, Minho Lee, and Young Ik Eom. 2017. Cffq: I/o scheduler for providing fairness and high performance in ssd devices. In *Proc. of IMCOM*. 1–6.

[42] Balgeun Yoo, Youjip Won, Seokhei Cho, Sooyong Kang, Jongmoo Choi, and Sungroh Yoon. 2011. SSD Characterization: From Energy Consumption's Perspective. In *Proc. of USENIX HotStorage*.