Improving Read Performance by Isolating Multiple Queues in NVMe SSDs

Minkyeong Lee
Sungkyunkwan University
Suwon 440-746, Korea
krong96@skku.edu

Dong Hyun Kang
Sungkyunkwan University
Suwon 440-746, Korea
kkangsu@skku.edu

Minho Lee
Sungkyunkwan University
Suwon 440-746, Korea
minhoz@skku.edu

Young Ik Eom
Sungkyunkwan University
Suwon 440-746, Korea
yieom@skku.edu

ABSTRACT

As NVMe SSDs have become viable, recent researches have focused on optimizing the performance of NVMe SSDs, which provide multiple I/O queues to maximize the I/O parallelism of flash-chip, while traditional operating systems are designed ordinarily for single queue storage, such as HDD and SATA SSD. Unfortunately, no prior works have considered the write interference while NVMe SSD still has the possibility of the interference. This interference is crucial in read-intensive environments because write requests negatively affect the latency of read requests. In this paper, we propose a novel queue isolation scheme that efficiently eliminates the *write interference* and improves the read performance by isolating read and write requests. Our experimental results clearly show that our scheme improves the read performance by up to 33% in heavy read workloads, compared to Baseline.

CCS Concepts

•**Software and its engineering** → **Input / output;** •**Hardware** → *Analysis and design of emerging devices and systems;*

Keywords

NVMe SSD; Multi-queue; Write interference; Read/write isolation

1. INTRODUCTION

The use of NVM Express Solid-State Drives (NVMe SSDs) is rapidly growing because they provide outstanding I/O performance compared to traditional SSDs which employ SAS or SATA interface [5]. NVMe is one of the optimized interface for PCIe SSDs, and it supports multiple I/O queues to maximize I/O parallelism on the flash chip-level [4].

Unfortunately, the performance benefits of NVMe SSD become limited by the legacy kernel I/O stack (e.g., block layer) because it has been optimized for single queue storage devices such as HDD and SATA SSD. Thus, several studies are widely performed in various layers to optimize the storage stack of the NVMe SSD [9, 12, 13, 15, 16]. Also, an user-level framework is recently proposed for managing NVMe SSD directly in the userspace [8]. However, all of these works do not focused on the *write interference*: a write request can have an adverse effect on the latency of read requests because each read request may have to wait for an in-flight write request to complete [6, 10, 11].

In order to demonstrate the impact of the *write interference* on NVMe SSDs, we measured the read performance with microbenchmark and found that the *write interference* significantly reduces the performance of a real NVMe SSD (we discuss the details of this result in Section 2). Based on this observation, we propose a novel queue scheme eliminating the *write interference* in NVMe SSDs. Our scheme efficiently improves the performance of read workloads by separating read and write requests.

In summary, the contributions of this paper include:

- We deeply understand the write interference, which degrades the performance of read workloads, based on a real NVMe SSD.
- We design a novel queue scheme that classify request queues into read and write queues. Also, our scheme provides transparency to the upper software layer so that it can be applied to traditional systems without major modifications.
- We implemented our scheme on Linux kernel and compared it with other schemes on a real NVMe SSD. Our experimental results show that our approach significantly improves read performance by 33% over Baseline.

The remainder of this paper is organized as follows. Section 2 explains the background and motivation. Section 3 presents the design of our approach and Section 4 shows the experimental results. Section 5 concludes the paper with a summary and future works.

IMCOM'17,January 05-07, 2017, Beppu, Japan
Copyright 2017 ACM. ISBN 978-1-4503-4888-1/17/01…$15.00
DOI: http://dx.doi.org/10.1145/3022227.3022262

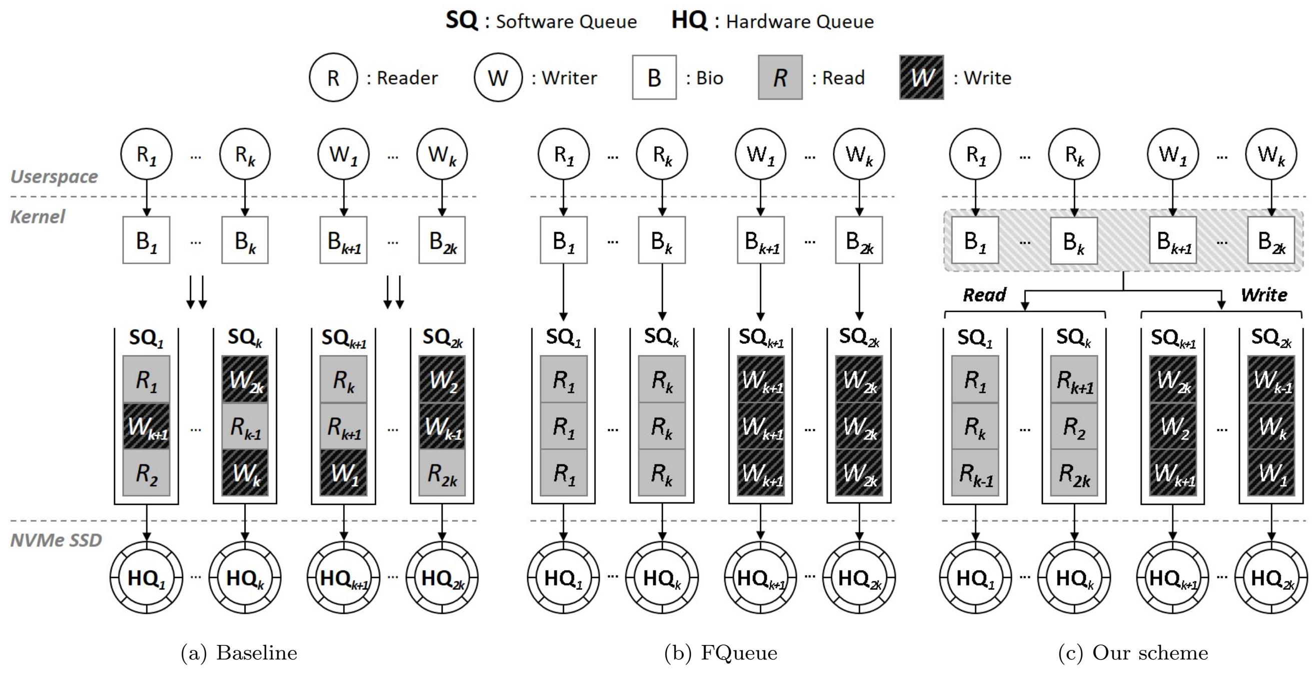


Figure 1: Software architectures of Baseline, FQueue and our scheme

2. BACKGROUND

In this section, we describe the software architecture for multiple queues. Moreover, we verify the negative effect of write interference in NVMe SSD through experiments.

2.1 Multi-queue Architecture

To understand the write interference on NVMe SSD, we study the software architecture of Linux kernel for NVMe SSD. The Linux kernel provides core-dedicated request queues called *software queues*, which help minimize the lock contention among the cores. Each software queue is directly linked with each submission queue in NVMe SSD (we call it *hardware queue*). In other words, a pair of software and hardware queue is dedicated to a core. For example, if a reader or writer runs on the *first* core, the kernel sends I/O requests issued from the reader or writer into the *first* software queue, and then the I/O requests are inserted into the *first* hardware queue.

Figure 1a illustrates a detailed process that I/O requests go through from userspace to NVMe SSD. We can classify this process into three steps: (1) A reader or writer submits an I/O request to the kernel in order to perform read or write operation; (2) The kernel converts the I/O request into `bio` and then inserts the `bio` into a core-dedicated software queue. For example, if a reader or writer is running on the first core, the kernel sends `bio` into the first software queue SQ_1. And if the process is migrated to the second core by process scheduling, the `bio` issued from the migrated process is inserted into SQ_2 instead of SQ_1 by the kernel. In this process, the kernel bypasses I/O scheduler. Thus, without ordering and merging, the `bio`s are inserted into software queue in a FIFO manner; (3) Finally, the kernel sends the `bio` into each SQ-dedicated hardware queue. Then, the NVMe controller of NVMe SSD processes the requests from the first hardware queue to the last in a round-robin manner.

2.2 Write Interference

The write interference occurs when read and write requests are mixed in software/hardware queues. As shown in Figure 1a, the kernel inserts both W_{k+1} and R_1 into the hardware queue HQ_1. HQ_1 receives W_{k+1} first and then R_1 secondly from SQ_1 (i.e., W_{k+1} is put in front of R_1 in HQ_1). As a result, it causes a delay in completing R_1 because R_1 has to wait for the completion of W_{k+1} [6, 10, 11].

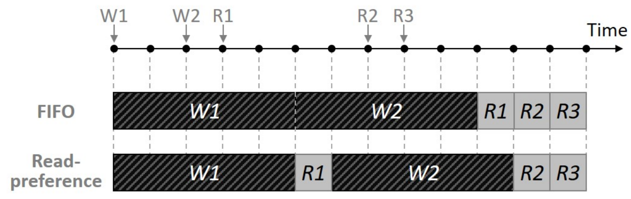
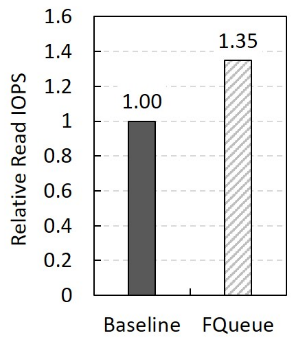
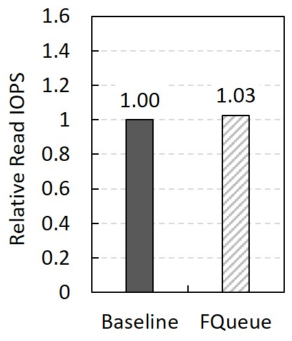


Figure 2: An example of FIFO and read-preference scheduling

To reduce the write interference, previous researchers have focused on I/O scheduling, and then they have proposed read-preference scheduling scheme that raise the priority of read requests. However, the read-preference scheduling still delays the latency of read requests because it does not completely eliminate the write interference in some cases [14]. For example, Figure 2 illustrates that $R2$ and $R3$ have no benefit from the read-preference scheduling even though the latency of $R1$ decreases. This is because both $R2$ and $R3$ must wait for the completion of $W2$. Moreover, Linux kernel



(a) Read performance when write interference is high



(b) Read performance when write interferece is low

Figure 3: Results of the microbenchmark to observe the effect of the write interference

bypasses I/O scheduler while using NVMe SSD. Therefore, the read-preference scheduling cannot apply to the system using NVMe SSD.

To verify the degradation of read performance caused by the write interference, we evaluated I/O performance of NVMe SSDs. We used Intel SSD 750 Series 400GB [3] as NVMe SSD. To generate I/O workloads, we deployed FIO benchmarking tool [2]. It was configured to simulate the read-intensive workloads with six random read threads (readers) and two random write threads (writers). The reason for this configuration is that we focus on read-intensive environments. The readers and writers generate burst I/O requests while either reading or writing 10GB files. We describe more detailed descriptions of the experimental setup in Section 4.

We compared Baseline with Fixed Queue (FQueue) where a reader or writer runs on only one core unlike Baseline. FQueue pins either reader or writer to a core by executing `taskset` command in order to separate read and write requests completely in software/hardware queues. Let us describe an example of FQueue by using Figure 1b. If a reader is running on the first core, not only SQ_1 but also HQ_1 will contain solely read requests as shown in Figure 1b. And if a writer is running on the $(k+1)$th core, both SQ_{k+1} and HQ_{k+1} will hold only write requests. This is because the reader and writer use only core-dedicated software/hardware queues. For this reason, FQueue does not have the write interference.

Figure 3a shows that, on average, random readers of FQueue have 35% higher I/O operations per second (IOPS) than that of Baseline. This is because the write interference is eliminated by isolating read and write queues. Thus, this result verifies that the write interference has a negative effect on the read performance in NVMe SSD. To better understand the effect of the write interference, we performed an additional experiment with setting I/O interval of random writers to 1ms. This setting means that the writers generate a few write requests. So, write requests less interfere read requests in a request queue. On the other hand, we did not change the I/O interval of random readers. In case where the write interference is low, Figure 3b shows that the read performance of FQueue is similar to that of Baseline. Therefore, we clearly confirm that as the write interference decreases, the read performance increases in NVMe SSD.

However, FQueue does not guarantee *fairness of performance* among the threads because FQueue favors some threads disproportionately (we specifically describe it in Section 4). Consequently, it is necessary to consider a practical alternative which not only isolates read and write queues but also maintains stable performance of threads. This observation gives the motivation of our design.

3. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of our scheme. We aim to achieve three goals:

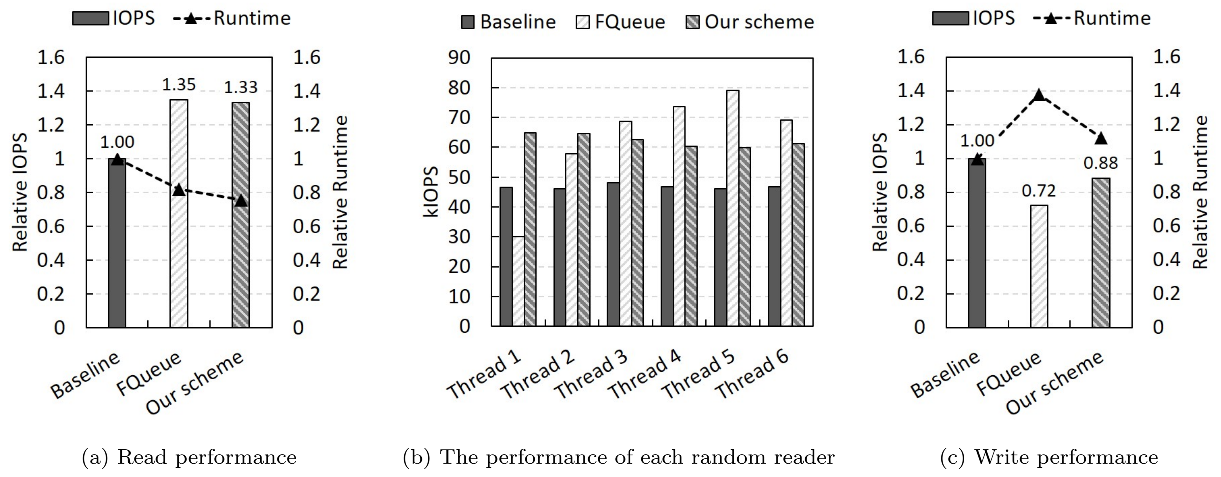
- Isolate read and write queues in the block layer.
- Improve read performance by eliminating the write interference.
- Guarantee stable performance of read workloads.

Figure 1c depicts the software architecture of our scheme. In our scheme, `bio` is examined to check whether it is a write request or not, unlike traditional way that checks current core number. After examining the `bio`, it is inserted into the corresponding software queue. For example, if B_k is a write request, it is sent into a write queue such as SQ_{k+1} instead of SQ_1 or SQ_k which is a read queue. And if B_k is a read request, it is issued to one of the read queues such as SQ_1 and SQ_k. Due to this queue dedication, read requests are completely separated from write requests. Subsequent to this process, `bio` in the software queue is inserted into the devoted hardware queue that is mapped one-to-one to the software queue similar to the traditional way. For instance, Figure 1c shows that HQ_1 receives read requests (e.g., R_{k-1}, R_k and R_1) from SQ_1. As a result, HQ_1 processes only read requests from SQ_1 without any disturbance of write requests.

Algorithm 1 Basic algorithm of our scheme

```
Input  : q: request queue, rw: read-write flag of request
Output: ctx: software queue
 1: unsigned int q_orig = get_cpu();
 2: unsigned int q_new = 0;
 3: if q->nr_hw_queues > 2 then
 4:     if rw == WRITE then
 5:             /* WQn is a write queue number */
 6:         q_new = (unsigned int)(WQn);
 7:     else    /* There is a read request */
 8:             /* RQn is a read queue number */
 9:         q_new = (unsigned int)(RQn);
10:     end if
11:     return __blk_mq_get_ctx(q, q_new);
12: else
13:     return __blk_mq_get_ctx(q, q_orig);
14: end if
```

Algorithm 1 outlines the procedure of our scheme for determining a software queue based on the characteristics of requests. First, this algorithm checks whether the device



(a) Read performance (b) The performance of each random reader (c) Write performance

Figure 4: Comparison of performance among Baseline, FQueue, and our scheme when the ratio of read queues to write queues is 3 to 1 (R:W=RQ:WQ=3:1)

provides multiple queues by performing `q->nr_hw_queues > 2` (Line 3). If the device does not provide multiple queues, a request is handled with the existing method that uses the core number as queue number (Line 1, 13). Otherwise, our scheme examines whether a request is write or not by checking the read-write flag of the request (Line 4). If the result of this operation is true, it is a write request, and otherwise, it is a read request. When the request is write, our scheme assigns write queue number `WQn` to `q_new` (Line 6). On the other hand, when the request is read, `q_new` is assinged to read queue number `RQn` (Line 9). Finally, our scheme performs `__blk_mq_get_ctx()` function with `q_new` that has either read or write queue number for getting a dedicated queue for read or write (Line 11). The queue retured from this function is used as software queues.

Meanwhile, requests may still get mixed at the flash controller, although requests are separately placed into queues. However, our scheme reduces this overhead by placing read queues and write queues in a grouped pattern (i.e., a group of read queues are followed by another group of write queues). This architecture decreases the degree of request mixing, while the NVMe controller supports round-robin scheduling scheme. For example, let us assume that hardware queues from HQ_1 to HQ_6 are read queues, and both HQ_7 and HQ_8 are write queues. These hardware queues are serviced in a round-robin manner, from HQ_1 to HQ_8 in this case. In other words, flash controller executes read commands from HQ_1 to HQ_6, and then performs write commands from HQ_7 to HQ_8. This process is performed repeatedly while there are requests in the hardware queues. On the other hand, with existing scheme, read and write commands are mixed severely at the flash controller during the round-robin scheduling process.

As a result, our scheme not only efficiently isolates read and write requests without heavy overhead, but also reduces the degree of request mixing at the flash controller layer. In addition, we can guarantee stable performance of the threads because it does not dedicate a software queue to a core.

4. EVALUATION

In this section, we show the evaluation results to address the following questions.

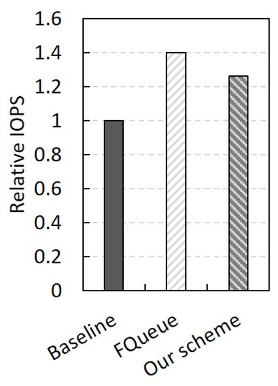
- Can the read performance be improved by eliminating the write interference?
- Is stable performance of threads guaranteed in our scheme?
- Can our scheme be applied to various workloads?

4.1 Experimental Setup

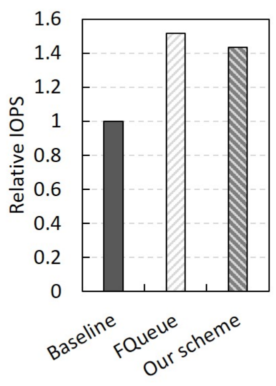
We used an 8-core server equipped with Intel Xeon E5506 processor and Ubuntu 12.04 LTS with Linux kernel 4.2.6 version as operating system. In addition, we deployed Intel SSD 750 Series 400GB as our NVMe SSD. Our experiments are based on FIO 2.2.12 version. Using FIO, we generated random readers and random writers which either read or write 10GB randomly. In this process, we referred to the ratios of webproxy and webserver workloads which are read-intensive workloads in Filebench benchmark tool [1]. Meanwhile, we measured the performance of random readers after writing 10GB files to NVMe SSD in advance to completely exclude the impact of writing. Moreover, we set `O_DIRECT` to eliminate the interference of buffer cache to I/O performance. Also, we set the value of *iodepth* to 32 in order to make burst I/O requests. The remainder of FIO options are set with default values.

4.2 Experimental Result

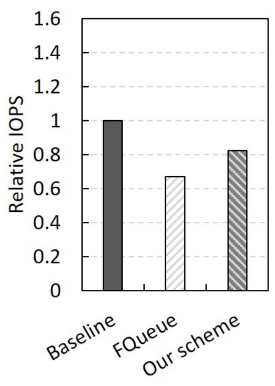
First, we generated six random readers and two random writers in order to build read-intensive environment (i.e., this workload consists of 75% read and 25% write requests). Also, we set the read-write queues ratio to 3:1 so as to process read-intensive workload enough. We compare the performance of random readers in order to verify that the write interference is eliminated. On average, our scheme achieves 33% higher read performance (IOPS) and 25% shorter runtime than Baseline as shown in Figure 4a. However, FQueue



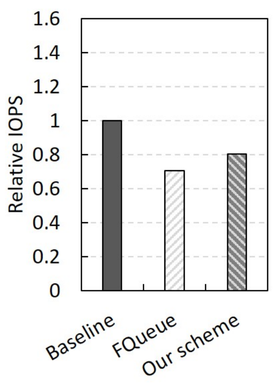
(a) Read performance (R:W=RQ:WQ=1:1)



(b) Read performance (R:W=RQ:WQ=1:3)



(c) Write performance (R:W=RQ:WQ=1:1)



(d) Write performance (R:W=RQ:WQ=1:3)

Figure 5: Comparison of performance among Baseline, FQueue, and our scheme by changing the ratio of read queues to write queues

has a little higher performance than our scheme because our scheme has the contention for read-write queues. Nevertheless, the read performance of our scheme is nearly equal to that of FQueue due to the elimination of the write interference, similar to FQueue. Unlike the results of IOPS, the runtime of our scheme is shorter than that of FQueue.

To better understand this result, we observe the performance of each random reader. Figure 4b illustrates that FQueue has unstable performance of random readers due to pinning threads to cores. This is because each core has unfair performance in the architecture of two quad-core processors. In contrast, our scheme has stable performance of random readers similarly to Baseline. This read performance has about 86% less standard deviation compared to that of FQueue. Therefore, we clearly confirm that the stability of read performance is guaranteed in our scheme.

Figure 4c shows the performance of random writers. Our scheme has 18% higher write performance on average than FQueue. This is because write requests issued from all of the cores are sent to write queues (i.e., the utilization of write queues increases). On the other hand, FQueue allows only some fixed cores to issue write requests to the write queues. Meanwhile, write IOPS of our scheme is a little bit lower than that of Baseline. However, this result is negligible because we focus on read performance for read-intensive workloads.

Finally, we compare the read and write performance by changing the ratio of read-write queues. When the ratio is 1 to 1 (the workload consists of 50% read and 50% write requests), read performance of our scheme is 26% higher than that of Baseline as shown in Figure 5a. Moreover, Figure 5b shows that our scheme has about 43% higher read IOPS compared to Baseline when the ratio of read-write queues is 1 to 3 (the workload consists of 25% read and 75% write requests). Therefore, we verified that our scheme can be applied to various workloads. Meanwhile, our scheme has a little lower read performance than FQueue due to lock contention while accessing software queues, and we leave it for future work.

In the case of write performance, our scheme has 22% and 14% higher write IOPS than FQueue, as illustrated in Figure 5c and 5d. Unfortunately, those write IOPSs are a little lower than those of Baseline. This is because our scheme can only use write queues during submission and completion of write requests, unlike Baseline which deploys all of the queues. But, since the improvement rate of read IOPS is much higher than that of write IOPS, consequently the overall performance increases by 8% on average. Moreover, we focus on improving read performance for read-intensive environments, and then it is a negligible result.

5. RELATED WORK

Some researchers have considered the write interference in flash-based SSDs. For instance, Chen et al. [6] analyzed the degradation of read performance caused by write request in SATA SSDs. They addressed that write operations induce asynchronous write-back and garbage collection in background, which has an adverse effect on foreground read operations. Moreover, another work attempted to shorten the read latency in mobile devices [10]. This work reduced the read latency by using I/O sorting that helps give higher priority to read requests in block layer. However, the distinctive difference between our work and these works is that we target on multi-queue SSDs such as NVMe SSDs.

Several works have focused on improving the I/O performance of NVMe SSD by deploying multiple queues. One of the works assigned a request queue per virtual machine and dedicated I/O threads which process I/Os asynchronously [9]. As a result, the work increased the I/O parallelism in virualized systems. In addition, Jun et al. [7] made a queue scheduler in self-virtualized SSD. They reduced the latency of virtual machines by balancing workloads across the request queues. But, our work differs from these works in that they failed to maximize the I/O performance of NVMe SSD sufficiently. This is because they never considered the write interference on NVMe SSD.

To directly manage NVMe SSD, recently, an user-level I/O framework is proposed [8]. It allows an user to control not only the number but also the purpose of queues with modifying NVMe driver and applications. So, an user may isolate read and write queues by deploying the framework. However, it has some drawbacks: security problems such as accessing unauthorized memory by a malware; mode switching overhead between user and kernel; and modifying user-level applications. On the other hand, our scheme never have these problems because we make the best use of the kernel sources.

6. CONCLUSION

No prior works have focused on the write interference in multiple queues of NVMe SSD. Moreover, studies on I/O scheduling can never reduce the write interference because NVMe bypasses the I/O scheduler.

This paper proposes a novel queue scheme based on NVMe SSD that isolates read and write queues in order to mitigate the write interference. Our experimental results show that the read performance of our scheme is 33% higher than that of Baseline by reducing the write interference. We believe that read-intensive systems can fully utilize our isolation scheme.

As a future work, we will perform the comprehensive evaluations of our scheme in various environments such as large-scale servers. Moreover, we have a plan to optimize the performance of our scheme by mitigating the lock contention for read-write queues. Finally, we will develop our scheme in the way to dynamically change the read-write queues ratio according to the characteristic of workloads.

7. ACKNOWLEDGEMENTS

This work was supported by ICT R&D program of MSIP/IITP. [R0126-15-1065, (SW StarLab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays]. Young Ik Eom is the corresponding author of this paper.

8. REFERENCES

[1] Filebench (File System Microbenchmarks). http://sourceforge.net/projects/filebench.

[2] FIO (Flexible IO Tester). http://git.kernel.dk/?p=fio.git;a=summary.

[3] Intel SSD 750 Series Product Specification. http://www.intel.com/content/www/us/en/solid-state-drives/ssd-750-spec.html.

[4] NVM Express Specification. http://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_Gold_20141209.pdf.

[5] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th ACM International Systems and Storage Conference*. SYSTOR'13, ACM, 2013.

[6] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proceedings of the 2009 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS'09, ACM, 2009.

[7] B. Jun and D. Shin. Workload-aware Budget Compensation Scheduling for NVMe Solid State Drives. In *Proceedings of the 4th IEEE Non-Volatile Memory System and Applications Symposium*. NVMSA'15, IEEE, 2015.

[8] H.-J. K. Kim, Y.-S. Lee, and J.-S. Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimizatino on NVMe SSDs. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems*. HotStorage'16, USENIX, 2016.

[9] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom. Improving Performance by Bridging the Semantic Gap between Multi-queue SSD and I/O Virtualization Framework. In *Proceedings of the 32nd IEEE International Conference on Massive Storage Systems and Technologies*. MSST'15, IEEE, 2015.

[10] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang. Reducing Smartphone Application Delay through Read/Write Isolation. In *Proceedings of the 13th ACM International Conference on Mobile Systems, Applications and Services*. MobiSys'15, ACM, 2015.

[11] 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'12, USENIX, 2012.

[12] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom. OS I/O Path Optimizations for Flash Solid-state Drives. In *Proceedings of the 2014 USENIX Annual Technical Conference*. ATC'14, USENIX, 2014.

[13] Y. Son, H. Kang, H. Han, and H. Y. Yeom. An Empirical Evaluation of NVM Express SSD. In *Proceedings of the 2015 IEEE International Conference on Cloud and Autonomic Computing*. ICCAC'15, IEEE, 2015.

[14] G. Wu and X. He. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. FAST'12, USENIX, 2012.

[15] Q. Xu, H. Siyamwala, M. Ghosh, M. Awasthi, T. Suri, Z. Guz, A. Shayesteh, and V. Balakrishnan. Performance Characterization of Hyperscale Applications on NVMe SSDs. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS'15, ACM, 2015.

[16] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. Performance Analysis of NVMe SSDs and their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR'15, ACM, 2015.