Article

Practical Enhancement of User Experience in NVMe SSDs

Seongmin Kim ¹ , Kyusik Kim ¹ , Heeyoung Shin ² and Taeseok Kim ^{3,*}

¹ Department of Computer Engineering, Kwangwoon University, 20, Gwangun-ro, Nowon-gu, Seoul 01897, Korea; smkim912@kw.ac.kr (S.K.); kks@kw.ac.kr (K.K.)

² Department of Intelligent System and Embedded Software Engineering, Kwangwoon University, 20, Gwangun-ro, Nowon-gu, Seoul 01897, Korea; hyshin@kw.ac.kr

³ School of Computer and Information Engineering, Kwangwoon University, 20, Gwangun-ro, Nowon-gu, Seoul 01897, Korea

* Correspondence: tskim@kw.ac.kr; Tel.: +82-2-940-5774

Received: 3 June 2020; Accepted: 9 July 2020; Published: 10 July 2020

Abstract: When processing I/O requests, the current Linux kernel does not adequately consider the urgency of user-centric tasks closely related to user experience. To solve this critical problem, we developed a practical method in this study to enhance user experience in a computing environment wherein non-volatile memory express (NVMe) solid-state drives (SSDs) serve as storage devices. In our proposed scheme, I/O requests that originate from the user-centric tasks were preferentially served across various levels of queues by modifying the multi-queue block I/O layer of the Linux kernel, considering the dispatch method of NVMe SSDs. Our scheme tries to give as fast a path as possible for I/O requests from user-centric tasks among many queues with different levels. Especially, when the SSD is overburdened, it avoids the queues with many pending I/O requests and thus can significantly reduce the I/O latency of user-centric tasks. We implemented our proposed scheme in the Linux kernel and performed practical evaluations on a commercial SSD. The experimental results showed that the proposed scheme achieved significant enhancement in the launch time of five widely used applications by up to ~65%.

Keywords: NVMe SSD; user experience; multi-queue block I/O layer

1. Introduction

Non-volatile memory express (NVMe) is an open logical device interface specification for accessing fast storage media, such as solid-state drives (SSDs) [1–4]. For SSDs to provide high-speed I/O, NVMe supports up to 64K submission and completion queues capable of queuing up to 64K commands [5–10]. Such a scalable architecture facilitates the full utilization of the internal parallelism of SSDs [11–16]. A multi-queue block I/O layer was introduced in the recent Linux kernel to efficiently support the NVMe SSDs in the host. This layer uses two levels of queues to improve scalability. One level is the software queues (SWQs) to alleviate the lock contention problem in multi-core environments, and the second level is the hardware queues (HWQs) to deal with some storages that support multiple dispatch queues, such as the NVMe SSDs [17–19].

In the previous single-queue block I/O layer, all I/O requests originating from tasks running on each CPU core were handled via a single request queue (Figure 1a). This resulted in a performance bottleneck due to the lock contention for accessing the single request queue [18,20]. In addition, it could not sufficiently exploit the potentiality of storages that support multiple dispatch queues. To alleviate these critical problems, the multi-queue block I/O layer employs two levels of queues (Figure 1b): SWQs and HWQs. The I/O requests originating from tasks running on any CPU core are first sent to

the corresponding SWQ mapped to the core. Thus, each CPU core is not required to consider the race condition caused by the access from other CPU cores. The I/O requests in the SWQ are subsequently moved to the HWQ, which is mapped to a single submission queue (SQ) and the said SWQ. The I/O requests are finally sent to the SQ, and the NVMe SSD retrieves them in a basic or weighted round-robin (WRR) fashion [17,21–24]. After being processed by the SSD device, completion messages for the I/O requests are inserted in the completion queue (CQ) grouped with the said SQ, and the insertion is notified to the host.

Figure 1. Type of block I/O layer: (a) Single-queue block I/O layer and (b) multi-queue block I/O layer.

This paper presents a practical scheme to enhance user experience by modifying the multi-queue block I/O layer. The main focus of this study is on the fast I/O handling of user-centric tasks, such as foreground or interactive tasks, due to their large impact on user experience [25]. The current Linux kernel does not adequately consider the urgency of the user-centric tasks, which especially issues I/O requests. To solve this problem, we first assigned higher process priorities to the user-centric tasks and identified the I/O requests originating from them in the block I/O layer. Subsequently, the structure of the multi-queue block I/O layer is modified to handle the aforementioned I/O requests as quickly as possible. Particularly, an NVMe feature that dispatches I/O requests from multiple SQs in a round-robin fashion is considered. The results obtained from various experiments demonstrated that the proposed scheme significantly enhances user experience from various perspectives.

The remainder of this paper is organized as follows. Related works and the details of our proposed scheme are presented in Sections 2 and 3, respectively. In Section 4, we present the evaluation results of our proposed scheme with an I/O benchmark tool—*fio*, flexible I/O tester—and five open source programs. The paper is then concluded in Section 5.

2. Related Works

Several studies on improving the multi-queue block I/O layer for NVMe SSDs have been reported in the literature. Joshi et al. [17] implemented a mechanism that supports four I/O priorities using two features. One feature is the I/O scheduling classes of Linux that consist of real-time, best-effort, none, and idle; the other feature is the WRR of the NVMe SSDs, a method for NVMe SSDs to retrieve more I/O requests from SQs with higher priorities. The authors increased the number of SQs in a

single set allocated to each CPU core from 1 to 4, and each SQ belonging to the set had one of the following SQ priorities; urgent, high, medium, and low. By mapping the I/O scheduling classes to the SQ priorities, differentiated I/O services were provided according to the I/O classes.

Lee et al. [21] solved a write interference problem, a situation in which the small number of write requests in a read-intensive workload negatively affects the performance of the workload. The problem was solved by splitting and inserting I/O requests into different SWQ according to the I/O type; that is, read or write. The I/O requests isolated in the different SWQ are also sent to different HWQ and SQ. This suggestion alleviates the write interference and consequently increases read performance by 33%.

Qian et al. [12] analyzed runtime behaviors in nonuniform memory access (NUMA) architecture consisting of multiple CPUs and NVMe SSDs in terms of I/O performance and energy efficiency. Based on this, the authors proposed an energy efficient I/O scheduler that manages I/O threads accessing NVMe SSDs, not only to reduce energy consumption and CPU usage, but also to guarantee I/O throughput and latency. Ahn et al. [26] also studied systems based on NUMA. The authors proposed an I/O resource management technique—weight-based dynamic throttling—to facilitate an efficient sharing of I/O resources in Linux cgroup on NUMA multi-core systems that use high-performance NVMe SSDs.

Kim et al. [27] solved a problem of the multi-queue block layer of the current Linux kernel being unable to reflect process priority when the process requests I/O operations sent to NVMe SSDs. The authors added additional queues between existing SWQs and HWQs to hold I/O requests that are issued by processes and lack opportunities—called to a token in their paper—to send the I/O requests to the NVMe SSDs at that point. Considering these works, it is clear that several studies have solved various problems, especially those caused by the structure of the current multi-queue block layer. However, no study has proposed an appropriate solution at the level of the Linux kernel for I/O-intensive and user-centric tasks to achieve more services, whether or not SSDs already process a large number of I/O requests issued by non-user-centric tasks.

3. Redesign of the Multi-Queue Block I/O Layer to Improve User Experience

This section describes the redesign of the multi-queue block I/O layer to improve user experience. The exclusive focus of the multi-queue block I/O layer on the I/O bandwidth may result in a bad user experience. To address this concern, this study aims at optimizing the I/O processing time of user-centric tasks by swiftly sending I/O requests issued by user-centric tasks to the SSD device through the complex multi-queue block I/O layer. We first assign a higher process priority to the user-centric tasks than non-user-centric or background tasks. When a program is launched for the first time or a program running in the background switches to the foreground, the user-centric tasks are automatically assigned a high process priority via a modified shell program. The modified shell can differentiate foreground and background tasks, and easily modify the process priority by using the *setpriority()* system calls. This facilitates a faster execution of user-centric tasks compared to non-user-centric ones through the task scheduling of a CPU scheduler, such as the completely fair scheduler. Consequently, a faster issuance of I/O requests from user-centric tasks to the multi-queue block I/O layer is achieved [28,29].

However, the current Linux kernel does not support any I/O service differentiated by process priority. Thus, this approach is still inadequate to preferentially process I/O requests issued by the user-centric tasks. Furthermore, process priority information disappears at the level of the block I/O layer by default. As the first step to process I/O requests from user-centric tasks preferentially, we passed the priority information to the block I/O layer by adding it to *bio* and *request*, which are basic structures used for I/O processing in the multi-queue block I/O layer. Once the I/O requests in the SWQ are passed to the SQ via the HWQ, the host loses control over them. To process I/O requests issued by user-centric tasks first, we divide the SWQ into two for every core: one queue is for I/O requests from user-centric tasks, and the other queue is for I/O requests from non-user-centric tasks. By referring the

priority information passed through *bio* and *request* structure, each I/O request is sent to the appropriate SWQ. If there are I/O requests in the SWQ for user-centric tasks, they are first moved to the HWQ.

I/O requests located in the HWQ are moved to the SQ immediately if there is sufficient space in the SQ. If there are other pending I/O requests in the HWQ or/and SQ, such as HWQ₂ and SQ₂ in Figure 2, the I/O requests from user-centric tasks, such as T_U in Figure 2, cannot be served until other I/O requests are retrieved from the queue holding them and processed. Moreover, NVMe SSDs typically dispatch I/O requests in multiple SQs in a round-robin fashion. Therefore, the I/O requests from user-centric tasks should be moved to HWQ and SQ with the smallest number of pending I/O requests to facilitate their processing in a minimal time. To this end, we first modified the NVMe device driver of the Linux kernel, as the current NVMe device driver cannot obtain the number of I/O requests pending in the SQ. We measure this number using two pieces of information for each SQ: a *head* that is recorded to SQ head pointer included in the CQ entry, and a *tail* managed by the NVMe device driver.

Figure 2. Traditional I/O processing for user-centric task (T_U) and non-user-centric task (T_{NU}).

Figure 3 demonstrates how latency can be improved in accordance with the proposed scheme. Suppose that a user-centric task T_U is running on CPU₂, and other tasks notated as T_{NU} are simultaneously running on other CPUs. In this example, a single I/O request issued by the T_U is passed to a separate software queue SWQ₂^U that is assigned to handle user-centric tasks and mapped to CPU₂. In the original kernel (Figure 2), an I/O request issued by T_U should wait until all I/O requests pending in the SWQ, HWQ, and SQ are processed. On the other hand, in our proposed scheme, the I/O request does not need to wait in SWQ₂^{NU}, as it uses SWQ₂^U dedicated for user-centric tasks. In addition, unlike the original kernel, it is migrated to HWQ₁ and SQ₁ instead of HWQ₂ and SQ₂ as SQ₁ has the least number of pending I/O requests. Consequently, the I/O request can be processed swiftly, compared to other I/O requests.

Figure 3. Proposed I/O processing for user-centric task (T_U) and non-user-centric task (T_{NU}).

As our scheme basically tries to process I/O requests issued by user-oriented tasks first, there will be concerns that I/O requests issued by non-user-oriented tasks might suffer from starvation if the SSD is heavily loaded. I/O requests from a user-centric task find and go to the HWQ and SQ with the shortest queue length each time, so I/O requests from non-user-centric tasks can go to other queues instead. In addition, as all SQs are dispatched in a round-robin or weighted round-robin way, pending I/O requests from non-user-centric tasks can be processed sometime. In summary, when the SSD drive is overburdened, especially, even if the user-centric task issues the I/O requests infinitely, the I/O requests issued by non-user-centric tasks may wait for a long time, but there would be no infinite waiting situation.

Details of the operations in our modified multi-queue block I/O layer are shown in Figure 4. After an I/O request from task running on n -th CPU reaches the block I/O layer, a bio that is a data structure for describing a single I/O operation is changed to a new request (r_n), or it is merged to already existing requests. The block I/O layer then determines whether r_n is requested by user-centric processes. If it is, an x -th SQ (SQ_x) that contains the smallest number of pending I/O requests is selected as a target SQ to insert an NVMe I/O command for r_n instead of the initially mapped SQ. This quick selection of the target SQ_x is achieved because the current kernel determines SWQ, HWQ, and SQ for r_n at this level by default. Subsequently, r_n is enqueued to the HWQ_x mapped to the SQ_x via n -th SWQ for user-centric processes (SWQ_n^U). After dequeuing from HWQ_x , in the level of the NVMe device driver, r_n is changed as a format of the NVMe I/O command and enqueued to the SQ_x if it is not full. The NVMe device driver finally notifies the insertion by updating a doorbell for the SQ_x . If r_n is requested by non-user-centric processes, it is enqueued to SWQ_n^{NU} . Before enqueueing r_n to the HWQ_n , it is confirmed whether there are pending I/O requests in the SWQ for the user-centric processes, SWQ_n^U . If they exist, the I/O requests located in SWQ_n^U are first dequeued from the SWQ and enqueued to HWQ_n to be processed before I/O requests from the non-user-centric processes. Owing to these various approaches, the modified block I/O layer can process I/O requests issued by the user-centric processes faster.

Figure 4. Flow chart of the proposed multi-queue block layer.

4. Performance Evaluation

The experimental environment is presented in Table 1. To emulate the I/O intensive applications, we used a *fio* benchmark tool that is widely used for generating I/O workloads with various configurations [30,31]. For enough I/O workloads, 50 *fio* tasks continuously generating random read requests were executed: one of them was set as a user-centric task, and the others were set as non-user-centric tasks. As mentioned earlier, the type of tasks is determined by the priority of each task. To verify the effectiveness of the ideas employed in the proposed scheme, performance evaluations were performed under the various combinations of ideas as described in Table 2. Note that we repeated all experiments 10 times to make the results reliable.

Figure 5 depicts the performance of the proposed scheme in terms of execution time, input/output operations per second (IOPS), and I/O bandwidth. This plot only shows the average value of the repeated experiments because the deviation is negligibly small. It can be observed that merely increasing the priority of user-centric tasks resulted in a significant performance boost (*high-priority*). In this case, the execution time, IOPS, and I/O bandwidth of user-centric tasks improved by 10.50%, 11.79%, and 11.49%, respectively, compared to the original kernel. In addition, when additional SQs were assigned to handle user-centric tasks (*separated*) and I/O requests from user-centric tasks were sent to the shortest submission queue (*shortest*), all metrics improved by up to 14.62% and 16.75%,

respectively. When all ideas were employed together (*proposed*), all metrics of user-centric tasks improved by up to 19.54%. Likewise, the use of the proposed scheme also seemed to improve the performance of non-user-centric tasks by up to 2.89%. It seems to be because, in our experiment environment, after a fio task executed as a user-centric task outputs the results and exits, fio tasks executed as non-user-centric tasks used the remaining resources. This behavior could be also observed in the average I/O latency, which was measured in the block I/O layer. The average latency of I/O requests from user-centric tasks is improved from 23.57us to 20.53us, while that of I/O requests from non-user-centric tasks is slightly improved from 23.57us to 22.20us.

Table 1. Experimental environment.

CPU	Intel i7-8700K CPU @ 3.70 GHz (6 cores)	
Storage	Samsung SSD 970 PRO	
	- Interface: PCIe 3.0 x 4, NVMe 1.3	- Capacity: 512GB
Operating system	Ubuntu 14.04 LTS 64-bit (Linux 4.13.10)	
Shell	Bash 4.4.18	
I/O workload	fio 3.6	
	- I/O engine: libaio	- LBA range: 15GiB
	- I/O pattern: random read	- Number of threads: 512

Table 2. Various evaluation targets.

Notation	Description
<i>original</i>	Evaluates with an unmodified kernel and shell program
<i>high-priority</i>	Raises the process priority of the user-centric tasks through the modified shell program
<i>separated</i>	Provides a separated software queue for user-centric tasks based on <i>high-priority</i>
<i>shortest</i>	Delivers I/O requests from user-centric tasks to the shortest-sized SQ based on <i>high-priority</i>
<i>proposed</i>	Includes all ideas: <i>high-priority</i> , <i>separated</i> , and <i>shortest</i>

Figure 6 depicts the launch times of five widely used Linux programs (Table 3), which is a crucial metric for users. As they are rather large programs using window, many files such as executables, configurations, and libraries should be read for the start-up. It inevitably entails a long latency and it can make users wait longer if the storage is overburdened. We measured the launch times while several *fio* workloads ran in the background. We can measure the start time of application by monitoring when *exec()* in shell is called, but it is not easy to clearly define and measure the completion of a launch. In our experiments, as all applications are using window, we measured the completion time of a launch by monitoring when window is created with *wmctrl*. Compared to using the original scheme, the use of the proposed scheme improved the launch times of the five programs on average by 28.42%, 48.71%, 63.54%, 59.83%, and 65.13%, respectively. As shown in the figure, the deviation of launch times is quite large depending on the system situation, but the proposed scheme consistently shows a significant performance improvement in all experiments. As the launch time can significantly affect user experience, these improvements are more substantial than the results of previous experiments, which were on accumulated performance improvement.

Figure 5. Evaluation results according to evaluation targets: (a) Execution time, (b) input/output operations per second (IOPS), and (c) I/O bandwidth.

Table 3. Target programs to measure launch time.

Program	Description
firefox	Open source web browser supporting multi-platform [32]
totem	Gnome's desktop movie player [33]
writer	LibreOffice's word processor [34]
calc	LibreOffice's spreadsheet program [34]
impress	LibreOffice's presentation program [34]

Figure 6. Launch times of applications when employing the proposed scheme.

5. Conclusions

This paper presents a scheme to enhance user experience in a computing environment using NVMe SSDs as storage devices. By assigning a higher priority to user-centric tasks and modifying the shell, multi-queue block I/O layer, and NVMe device driver, I/O requests issued by user-centric tasks can be preferentially serviced. The results of various experiments performed in this study reveal that the proposed scheme significantly improves user experience for user-centric tasks by assigning to them higher priority in terms of I/O processing. In the future, we will continue to study, on the operating system level, supports for improving user satisfaction by prioritizing the user-centric tasks. As we discussed the effect of process priority in the performance evaluation section, more exquisite CPU scheduling level support is very important to improve the response time of user-centric tasks in a multi-core environment. In addition, memory allocation for user-centric tasks can be delayed by non-user-centric tasks due to lock contention, so it should be also considered to optimize the response time. We believe that the synthetic analysis of the relationship among different layers for complicated environments using multi-core and storages with a great many queues is necessarily required and a cross-layer design considering all of them is still an open problem that should be continuously studied.

Author Contributions: Methodology, H.S., T.K., K.K., and S.K.; software, H.S.; formal analysis, S.K., K.K., and H.S.; investigation, S.K. and K.S.; writing—original draft preparation, S.K. and H.S.; writing—review and editing, T.K., K.K., and S.K.; visualization, S.K. and K.K.; supervision, T.K.; project administration, T.K.; funding acquisition, T.K. All authors have read and agreed to the published version of the manuscript.

Funding: The present research has been conducted by the Research Grant of Kwangwoon University in 2020. This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2020R1F1A1074676).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Eshghi, K.; Micheloni, R. SSD architecture and PCI express interface. In *Inside Solid State Drives (SSDs)*; Springer Series in Advanced Microelectronics; Springer: Dordrecht, The Netherlands, 2013; Volume 37, pp. 19–45. ISBN 978-94-007-5146-0.
2. NVM Express: NVM Express Overview. Available online: https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf (accessed on 5 March 2020).
3. Bjørling, M.; Gonzalez, J.; Bonnet, P. LightNVM: The Linux Open-Channel SSD Subsystem. In Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17), Santa Clara, CA, USA, 27 February–2 March 2017; pp. 359–374.

4. Zhang, J.; Donofrio, D.; Shalf, J.; Kandemir, M.T.; Jung, M. NVMMU: A Non-Volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In Proceedings of the International Conference on Parallel Architecture and Compilation (PACT), San Francisco, CA, USA, 18–22 October 2015; pp. 13–24. [\[CrossRef\]](#)
5. Zhang, J.; Kwon, M.; Gouk, D.; Koh, S.; Lee, C.; Alian, M.; Chun, M.; Kandemir, M.T.; Kim, N.S.; Kim, J.; et al. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 477–492.
6. NVM Express Base Specification Revision 1.3c. Available online: https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3c-2018.05.24-Ratified.pdf (accessed on 5 March 2020).
7. Peng, B.; Zhang, H.; Yao, J.; Dong, Y.; Xu, Y.; Guan, H. MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through. In Proceedings of the USENIX Annual Technical Conference (USENIX ATC 18), Boston, MA, USA, 11–13 July 2018; pp. 665–676.
8. Kim, S.; Yang, J.S. Optimized I/O Determinism for Emerging NVM-based NVMe SSD in an Enterprise System. In Proceedings of the 55th Annual Design Automation Conference, San Francisco, CA, USA, 24–28 June 2018; pp. 1–6. [\[CrossRef\]](#)
9. Kim, H.J.; Lee, Y.S.; Kim, J.S. NVMeDirect: A User-space I/O Framework for Application-Specific Optimization on NVMe SSDs. In Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16), Denver, CO, USA, 20–21 June 2016.
10. Xu, Q.; Siyamwala, H.; Ghosh, M.; Awasthi, M.; Suri, T.; Guz, Z.; Shayesteh, A.; Balakrishnan, V. Performance Characterization of Hyperscale Applications on NVMe SSDs. In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, NY, USA, 15–19 June 2015; pp. 473–474. [\[CrossRef\]](#)
11. Awad, A.; Kettering, B.; Solihin, Y. Non-Volatile Memory Host Controller Interface Performance Analysis in High-Performance I/O Systems. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; pp. 145–154. [\[CrossRef\]](#)
12. Qian, J.; Jiang, H.; Srisa-An, W.; Seth, S.; Skelton, S.; Moore, J. Energy-Efficient I/O Thread Schedulers for NVMe SSDs on NUMA. In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, Spain, 14–17 May 2017; pp. 569–578. [\[CrossRef\]](#)
13. Jun, B.; Shin, D. Workload-Aware Budget Compensation Scheduling for NVMe Solid State Drives. In Proceedings of the IEEE Non-Volatile Memory System and Applications Symposium (NVMSA), Hong Kong, China, 19–21 August 2015; pp. 1–6. [\[CrossRef\]](#)
14. Yang, Z.; Hoseinzadeh, M.; Wong, P.; Artoux, J.; Mayers, C.; Evans, D.T.; Bolt, R.T.; Bhimani, J.; Mi, N.; Swanson, S. H-NVMe: A Hybrid Framework of NVMe-Based Storage System in Cloud Computing Environment. In Proceedings of the IEEE 36th International Performance Computing and Communications Conference (IPCCC), San Diego, CA, USA, 10–12 December 2017; pp. 1–8. [\[CrossRef\]](#)
15. Kim, J.; Ahn, S.; La, K.; Chang, W. Improving I/O Performance of NVMe SSD on Virtual Machines. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 4–8 April 2016; pp. 1852–1857. [\[CrossRef\]](#)
16. Bhimani, J.; Yang, J.; Yang, Z.; Mi, N.; Xu, Q.; Awasthi, M.; Pandurangan, R.; Balakrishnan, V. Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs. In Proceedings of the IEEE 35th International Performance Computing and Communications Conference (IPCCC), Las Vegas, NV, USA, 9–11 December 2016; pp. 1–8. [\[CrossRef\]](#)
17. Joshi, K.; Yadav, K.; Choudhary, P. Enabling NVMe WRR Support in Linux Block Layer. In Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17), Santa Clara, CA, USA, 10–11 July 2017.
18. Björling, M.; Axboe, J.; Nellans, D.; Bonnet, P. Linux block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In Proceedings of the 6th International Systems and Storage Conference, Haifa, Israel, 30 June–2 July 2013; pp. 1–10. [\[CrossRef\]](#)
19. Tavakkol, A.; Gómez-Luna, J.; Sadrosadati, M.; Ghose, S.; Mutlu, O. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST 18), Oakland, CA, USA, 12–15 February 2018; pp. 49–66.

20. Kim, T.Y.; Kang, D.H.; Lee, D.; Eom, Y.I. Improving Performance by Bridging the Semantic Gap Between Multi-Queue SSD and I/O Virtualization Framework. In Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST), Santa Clara, CA, USA, 30 May–5 June 2015; pp. 1–11. [[CrossRef](#)]
21. Lee, M.; Kang, D.H.; Lee, M.; Eom, Y.I. Improving Read Performance by Isolating Multiple Queues in NVMe SSDs. In Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication, Beppu, Japan, 5–7 January 2017; pp. 1–6.
22. Yang, T.; Huang, P.; Zhang, W.; Wu, H.; Lin, L. CARS: A Multi-layer Conflict-Aware Request Scheduler for NVMe SSDs. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 25–29 March 2019; pp. 1293–1296. [[CrossRef](#)]
23. Guignani, S.; Lu, X.; Panda, D.K. Analyzing, Modeling, and Provisioning QoS for NVMe SSDs. In Proceedings of the IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC), Zurich, Switzerland, 17–20 December 2018; pp. 247–256. [[CrossRef](#)]
24. Huang, S.M.; Chang, L.P. Providing SLO compliance on NVMe SSDs through parallelism reservation. *ACM Transact. Des. Autom. Electron. Syst.* **2018**, *23*, 1–26. [[CrossRef](#)]
25. Hahn, S.S.; Lee, S.; Yee, I.; Ryu, D.; Kim, J. Improving User Experience of Android Smartphones Using Foreground App-Aware I/O Management. In Proceedings of the 8th Asia-Pacific Workshop on Systems, Mumbai, India, 2–3 September 2017; pp. 1–8. [[CrossRef](#)]
26. Ahn, S.; La, K.; Kim, J. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems. In Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16), Denver, CO, USA, 20–21 June 2016.
27. Kim, K.; Hong, S.; Kim, T. Supporting the Priorities in the Multi-queue Block I/O Layer for NVMe SSDs. *J. Semicond. Technol. Sci.* **2020**, *20*, 55–62. [[CrossRef](#)]
28. Zhuravlev, S.; Saez, J.C.; Blagodurov, S.; Fedorova, A.; Prieto, M. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.* **2012**, *45*, 1–28. [[CrossRef](#)]
29. Wong, C.S.; Tan, I.K.T.; Kumari, R.D.; Lam, J.W.; Fun, W. Fairness and Interactive Performance of O (1) and CFS Linux Kernel Schedulers. In Proceedings of the International Symposium on Information Technology, Kuala Lumpur, Malaysia, 26–28 August 2008; pp. 1–8. [[CrossRef](#)]
30. Flexible I/O Tester. Available online: <https://github.com/axboe/fio> (accessed on 5 March 2020).
31. Son, Y.; Kang, H.; Han, H.; Yeom, H.Y. An Empirical Evaluation of NVM Express SSD. In Proceedings of the International Conference on Cloud and Autonomic Computing, Boston, MA, USA, 21–25 September 2015; pp. 275–282. [[CrossRef](#)]
32. Firefox. Available online: <https://www.mozilla.org/en-US/firefox/> (accessed on 5 March 2020).
33. App/Videos—GNOME Wiki. Available online: <https://wiki.gnome.org/Apps/Videos> (accessed on 5 March 2020).
34. LibreOffice—Free Office Suite. Available online: <https://www.libreoffice.org/> (accessed on 5 March 2020).

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).