The *Minimal-Effort Write* I/O Scheduler for Flash-based Storage Devices

Daekyu Park [†‡], Dong Hyun Kang[†], Seung Min Ahn[†], and Young Ik Eom[†]
[†]Sungkyunkwan University, Korea [‡]Samsung Electronics, Korea
{dekay.park,kkangsu,smahn9123,yieom}@skku.edu

***Abstract*--Unfortunately, current I/O stacks in the operating system cause the *read-blocked-by-write* problem because they try to issue write requests that came from the applications in a burst way. In this paper, we propose a novel I/O scheduler to efficiently mitigate the problem. The key idea of our scheduler is the *minimal-effort write* that calibrates the number of dispatched requests from the queues of the I/O scheduler. The evaluation results clearly show that our scheme improves overall read latency while guaranteeing the throughput required by the application. In the best case, our scheme reduces average read latency by up to 66% compared to conventional I/O schedulers.**

I. Introduction

Recently, home cloud systems are widely used and the number of smart devices (e.g., IoT and mobile devices) connected to the home cloud system is also increasing [1]. Unfortunately, several services on the home cloud system require a large amount of data and control, resulting in heavy burden on the system [2]. As generally known, the *quality-of-service* (QoS) of each service is closely affected by the latency of I/Os that are requested by the service. Therefore, the storage performance of the system is very important for the QoS control. As a result, conventional hard disk drives (HDDs) as a main storage of home cloud system, are gradually being replaced by flash-based storage devices, such as solid state drives (SSDs), which guarantee high-performance and low latency. Unfortunately, the existing I/O stacks in OS, such as page cache, file system, and block layer, have only been studied on HDD-based storage systems for a few decades. Thus, traditional I/O schedulers in the block layer (e.g., deadline, noop, anticipatory, and cfq), incurs inefficiency on flash-based storage systems [3].

Flash-based storage devices have radically different characteristics (e.g., asymmetric read/write speed, erase-before-write, and wear-out) compared with HDDs which have mechanical features. Especially, due to the asymmetric read and write speed, the read may be interfered by the relatively slower write when read and write requests are simultaneously issued to the underlying storage devices. Furthermore, the more bursts the write request, the worse the read throughput and latency. This has two reasons. First, if most of the resources (e.g., NCQ) have been already allocated to previous burst writes, read requests cannot be granted with the required resources. Second, since a single NAND chip serializes its operations such as read, program, and erase, the read operations issued from the host may be delayed during program execution.

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (NRF-2015M3C4A7065696)

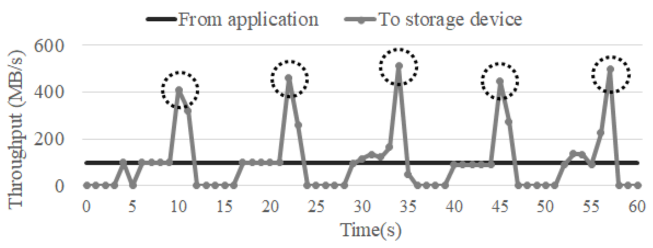


Fig. 1. Throughput with *write-back* policy

In this paper, we propose a novel I/O scheduler that throttles write requests slowly enough to efficiently solve the first issue. Our scheme reduces average read latency by up to 66% compared with the traditional approaches such as noop in Linux. Moreover, we confirmed that our scheme substantially reduces the overall long-tail latencies.

II. Background and Motivation

A. Write-back

Most operating systems support buffer cache with *write-back* policy to improve I/O performance [4]. The *write-back* policy temporarily stores data of write requests in the buffer cache without incurring the storage I/O. This policy helps to guarantee low latency and high throughput for the writes issued by write-intensive applications. But, the data on the buffer cache should be eventually evicted to reclaim free space. The eviction operation is triggered by the following conditions to reflect the up-to-date data to the underlying storage: exhaustion of free space in the buffer cache, expiration of pre-defined threshold time (e.g., pdflush daemon), and explicit issuance of synchronous operation (e.g., fsync, msync, and fdatasync). Unfortunately, at that time, the data should be flushed to the storage in a burst way.

Figure 1 shows the application-level and actual storage-level throughput during 60 seconds. In this experiment, an application continuously issues write requests to the underlying kernel layer while maintaining 100MB/s throughput. However, as mentioned before, the kernel does not reflect data of the write requests immediately to the storage underneath. Whenever the eviction is triggered with the above-mentioned conditions, the data on the buffer cache is explosively issued to the storage, occasionally in above 400MB/s.

B. Native command queuing (NCQ)

SATA II interface protocol for storage device supports *native command queuing* (NCQ) feature for the request queuing mechanism. NCQ was originally designed to reduce the seek

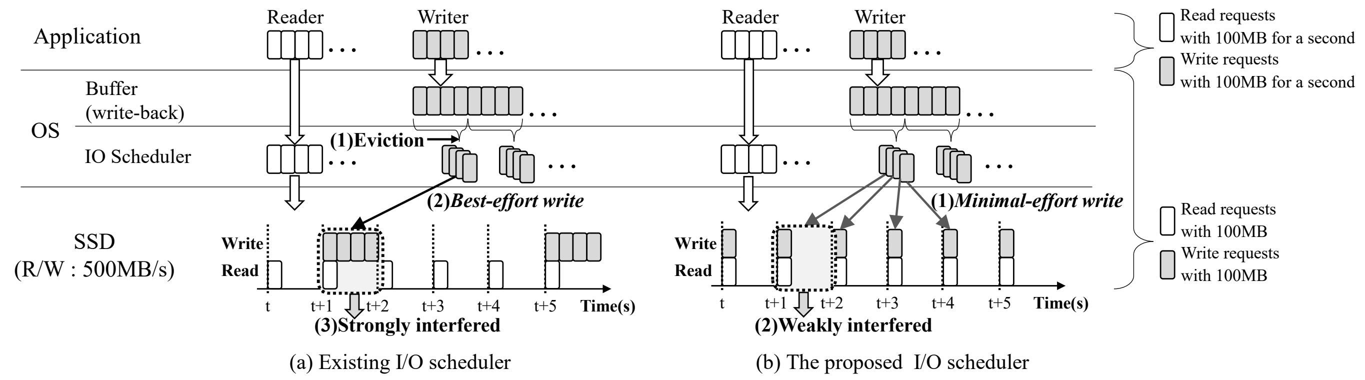


Fig. 2. Existing I/O scheduler (*best-effort write*) vs. the proposed I/O scheduler (*minimal-effort write*)

time of HDDs. However, it still has positive effects on the SSDs even though the devices have no mechanical parts. Since current flash-based storage device is made up of multi-flash packages, I/Os queued by NCQ exploit parallelism in core, channel, chip, and plane-level [5]. Therefore, as the number of queued I/Os (i.e., NCQ depth) inside the SSD increases, the performance of the storage is improved by maximizing the parallelism.

C. *Motivation*

In this section, we describe the relationship between the *write-back* policy and NCQ in terms of read and write interference. Figure 2a shows the example of *read-blocked-by-write* along with the traditional I/O scheduler. In this example, we assumed that both a reader and a writer issue 100MB requests per second (i.e., 100MB/s throughput), and the storage can handle 500MB read/write requests per second (i.e., 500MB/s throughput). (1) As mentioned before, write requests are temporally stored on the buffer cache and the data of the requests is flushed to the underlying storage, when any of the above-mentioned conditions is met. (2) At the time of flushing, the I/O scheduler tries to issue write requests from the buffer cache as many as possible. We call this the *best-effort write*. Unfortunately, the *best-effort write* has a huge negative effect on the latency of read request because it causes the burst writes to accommodate almost all resources of NCQ. (3) In summary, read requests are strongly interfered by write requests. This situation motivated us to re-design the I/O scheduler.

III. Design and Implementation

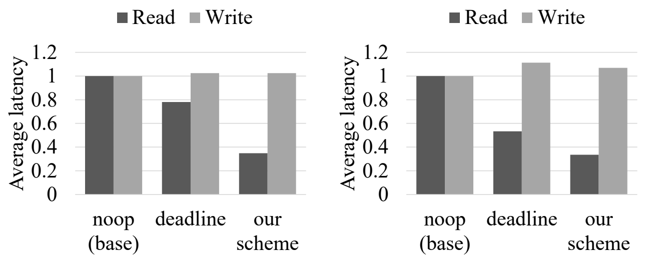
The goal of our work is to mitigate the *read-blocked-by-write* problem. To achieve our goal, we design a novel I/O scheduler that tries to issue write requests as lazy as possible. We call this the *minimal-effort write*. For the *minimal-effort write* scheme, we extend the noop I/O scheduler and added two key features: *queue depth calibration* and *read preference*.

Queue depth calibration: For queue depth calibration, we first classify the queues of the I/O scheduler into read queues and write queues, and then calibrate the number of dispatched writes from the write queues. For dynamic calibration, we measure the throughput handled by the I/O scheduler on-the-fly and compare it with the throughput that was set by application requirement. Then, if the throughput of the I/O scheduler is higher than the throughput set by the application, we gradually reduce the number of dispatched write requests until both throughputs become equal. As a result, we can minimize the *read-blocked-by-write* problem by throttling the write throughput. Figure 2b shows the overview of our I/O scheduler. If data is transferred from the buffer cache to our I/O scheduler by flush operation, our scheme throttles the amount of data to reach only the throughput of the application. (1) Therefore, unlike the existing I/O scheduler, our scheme issues the requests on the write queue by the *minimal-effort write* policy. (2) In conclusion, it causes a little negative effect on the latency of read requests (i.e., weakly interfered).

Read preference: To help the *queue depth calibration*, we also use the read preference scheme that prefers the read requests over the write requests in terms of dispatch on the I/O scheduler. This is because read request can be delayed by previous write requests queued in NCQ and this delay significantly degrades the latency of the read requests. For read preference, we set higher priority to read requests than write requests by using the *read/write ratio*. The ratio means how many read requests are issued to the storage before one write request. For example, if *read/write ratio* is set to 2:1, two read requests are dispatched from the read queue prior to each dispatch of write request. We also calibrate the *read/write ratio* by using the similar way to the queue depth calibration. Our scheme maximizes the *read/write ratio* while maintaining only the throughput required by the application, so it does not affect the performance in terms of applications.

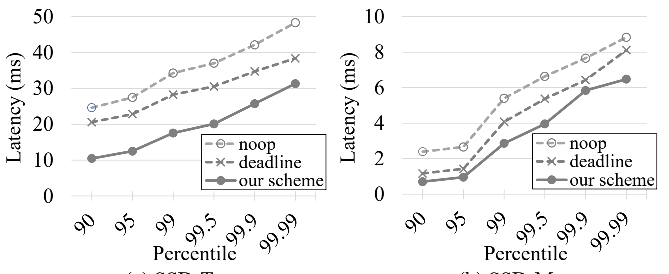
IV. Performance Evaluation

In this section, we describe our experimental setup and show the evaluation results, compared with those of the I/O schedulers supported in the current Linux kernel [6]. We turned off the merge operation on our scheme because it disturbs the *minimal-effort write* by making I/O requests too large. For a fair comparison, other I/O schedulers are also set to turn off the merge operation. We conducted our evaluation on a machine that ran Ubuntu 16.04 LTS (kernel version 4.4) with 3.40GHz Intel Core i7-6700 CPU with 8GB RAM. We employed two commercial SSDs with different NAND flash types,



(a) SSD-T (b) SSD-M

Fig. 3. Average latency (normalized to noop)



(a) SSD-T (b) SSD-M

Fig. 4. Tail latencies (90 to 99.99 percentile)

because the impact on *read-blocked-by-write* varies depending on the write performance of NAND flash type; one is an MLC flash-based SSD with 500MB/s write (SSD-M) and the other is a TLC flash-based SSD with 150MB/s write (SSD-T). For synthetic evaluation, we built a benchmark tool that consists of a reader and a writer task. Each task continuously generates read or write requests to maintain 100MB/s throughput. However, we configured different payload of the request for the reader and writer: 128KB for the writer and 4KB for the reader. We used the fact that small read requests show the impact of the *read-blocked-by-write* problem more clearly than large read requests.

Figure 3 shows the average latency of the requests and the average latency is normalized to that of noop I/O scheduler. As we expected, write latency of our scheme is comparable or slightly worse than that of conventional I/O schedulers. However, read latency of our scheme is significantly improved in all cases. Especially, as shown in Figure 3a, our scheme improves the average read latency by 56% and 44% compared to noop and deadline, respectively. In case of SSD-M (Figure 3b), our scheme also achieves 66% and 36% improvement in the average read latency compared to noop and deadline, respectively. The reason behind these results is that our scheme efficiently mitigates the *read-blocked-by-write* problem by throttling the write requests. To confirm the QoS in detail, we show the latency distribution for I/O operations (i.e., from 90 to 99.99 percentile tail-latencies) in Figure 4. As shown in Figure 4, our scheme shows better latencies in all cases. Especially, in case of SSD-M, our scheme improves by up to 70% compared with noop. In summary, our scheme tries to solve the *read-blocked-by-write* problem with *minimal-effort write* scheme. As a result, it improves the overall read latency while guaranteeing the write throughput required by the application.

V. Related Work

Several works focused on the *read-blocked-by-write* problem in terms of fairness for requests based on flash-based storage devices. Park *et al.* and Shen *et al.* studied the fairness of I/O scheduler and proposed FIOS [7] and FlashFQ [8], respectively, as their I/O schedulers. FIOS focuses on the processing time of different request sizes inside the storage and calibrates the number of issued I/O requests in accordance with time. FlashFQ extends FIOS to consider I/O intensive tasks. Meanwhile, some researchers [9], [10] studied the serialized operations inside the storage: chip-level and channel-level. Other researchers [11] modified the firmware of the storage to improve the latency. However, previous works consider only the features of the storage underneath without the application requirements. Therefore, they may cause run-time fluctuation on the performance in terms of the applications. On the other hand, our scheme guarantees the stable performance along with the application requirements.

VI. Conclusion

In this paper, we studied the *read-blocked-by-write* problem caused by traditional I/O stacks in the Linux kernel. To solve the problem, we proposed a novel I/O scheduler that improves the read latency by throttling the number of write requests while keeping the throughput required by the applications. Our scheme does not handle the process of merging I/O requests. To verify our improvements on various workloads, we plan to extend our scheme for I/O merge process and evaluate various request sizes including real-world workloads.

References

[1] C. Salzmann and D. Gillet, "Smart device paradigm standardization for online labs," in Proc. of the IEEE EDUCON, pp. 1217-1221, 2013.

[2] H. Gu, Y. Diao, W. Liu, and X. Zhang, "The design of smart home platform based on cloud computing," in Proc. of the IEEE EMEIT, vol. 8, pp. 3919-3922, 2011.

[3] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. Noh, "Disk schedulers for solid state drivers," in Proc. of the ACM EMSOFT, pp. 295-304, 2009.

[4] R. Kardela, S. Love, and B.G. Wherry, "Caching strategies to improve disk system performance," IEEE Trans. Computer, vol. 27, no. 3, pp. 38-46, 1994.

[5] J. Kang, J. Kim, C. Park, H. Park, and J. Lee, "A multi-channel architecture for high-performance NAND flash-based storage system," Journal of Systems Architecture, vol. 53, no. 9, pp. 644-658, 2007.

[6] D.P. Bovet and M. Cesati, *Understanding the Linux Kernel, 3-ed., O'Reilly*, pp. 572-585, 2006.

[7] S. Park and K. Shen, "FIOS: A fair, efficient flash I/O scheduler," in Proc. of the USENIX FAST, pp. 13-13, 2012.

[8] K. Shen and S. Park, "FlashFQ: A fair queueing I/O scheduler for flash-based SSDs," in Proc. of the USENIX ATC, pp. 67-78, 2013.

[9] C. Gao, L. Shi, M. Zhao, C. Xue, K. Wu, and E. Sha, "Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives," in Proc. of the IEEE MSST, pp. 1-11, 2014.

[10] Q. Zhang, D. Feng, F. Wang, and Y. Xie, "An efficient, QoS-aware I/O scheduler for solid state drive," in Proc. of the IEEE HPCC, pp. 1408-1415, 2013.

[11] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. Kandemir, C. Das, and M. Jung, "Exploiting intra-request slack to improve SSD performance," in Proc. of the ACM ASPLOS, pp. 375-388, 2017.