IOPriority: To The Device and Beyond

Adam Manzanares, Filip Blagojević, Cyril Guyot
Western Digital Research

Abstract

In large scale data centers, controlling tail latencies of IO requests keeps storage performance bounded and predictable, which is critical for infrastructure resource planning. This work provides a transparent mechanism for applications to pass prioritized IO commands to storage devices. As a consequence, we observe much shorter tail latencies for prioritized IO while impacting non-prioritized IO in a reasonable manner. We also provide a detailed description of the changes we made to the Linux Kernel that enable applications to pass IO priorities to a storage device. Our results show that passing priorities to the storage device is capable of decreasing tail latencies by a factor of 10x while decreasing IOPS minimally.

1 Introduction

Hard disk drives (HDD) have been a part of the computer storage hierarchy since the 1950s. Although alternative technologies (such as flash) have been introduced, HDDs are still relevant to this day due to their capacity, performance, and cost properties, which place them firmly between tape and flash. HDD technology improvements have pushed the devices to greater densities, but the fact remains that the HDD is a mechanical device with limited opportunities for request parallelism. This has led to a decrease in the performance to capacity ratio of HDDs, measured in IOPS/TB, due to the fact that capacity has increased while IOPS have remained nearly flat.

In order to increase the IOPS from a single actuator arm that controls HDD heads, modern hard disk drives employ device level queues that are managed by drive firmware schedulers. These schedulers leverage drive internal information, such as magnetic head position, to determine the IO request that should be served next. Although these sophisticated schedulers are able to improve the throughput of a set of requests, this comes at a cost of increased tail latencies. Furthermore, a mechanism that would preserve the IO order between the drive and the hostside IO queues is currently non-existent. As a consequence, host-drive IO *re-scheduling* is a common occurrence, i.e. IO requests ordering established at the host side experience reordering once the requests reach the drive. A common effect caused by *re-scheduling* are unpredictable and long IO latencies. At the same time, with the rise in cloud based storage, and the fact that HDDs are the central element of this storage, HDD tail latency has become a critical performance factor [7].

To limit the worst case latency for performance-critical IO requests, two main approaches have been considered in production environments: (i) prioritizing the "real-time" over the "background" IO requests, and (ii) limiting the number of IO requests issued to a storage device. Although the first option appears to be a reasonable approach it is rarely used in production settings due to the host's inability to communicate the priority information to a storage device. Consequently any high priority IO request is likely to get rescheduled by the device scheduler, and possibly experience high delays. Currently, limiting the number of issued IO requests is the default option in many data centers. Submitting a low number of IOs to a storage device limits the effects of the device IO scheduler and reduces IO tail latencies. Unfortunately, the low number of outstanding IOs also prevents users from reaching the highest possible IO throughput.

In this work we provide a path of communication between a user application and a storage device that allows device-level IO request prioritization. Passing the application-level priorities to a storage device scheduler minimizes IO *re-scheduling*. Additionally this reduces the tail latency of performance-critical IOs without throttling IO requests and disabling the device scheduler. The communication path now includes user applications, Linux block schedulers, and the drive scheduler on the other end. It is important to note that our work leverages existing IO prioritization APIs without any changes and our implementation has been merged into the 4.10 series kernel [4]. The rest of this paper includes a discussion of the host and drive level queue interactions as well as a set

of results that stress the effectiveness of bridging the host and HDD schedulers. Although our work was motivated by HDDs, our solution is relevant to applications leveraging storage devices that employ internal queueing, such as solid-state drives and future storage-class memories.

2 Host-Device Queue Interaction

Adding support for priority being passed to storage devices is a critical tuning knob when devices queue commands internally. The SATA, SCSI, and NVMe storage standards have support for queued commands and the storage device is allowed to reorder or delay these queued commands based on the device internal information that is not visible to the host. Typically HDD schedulers have favored throughput over latency, which is no longer prudent for cloud storage systems.

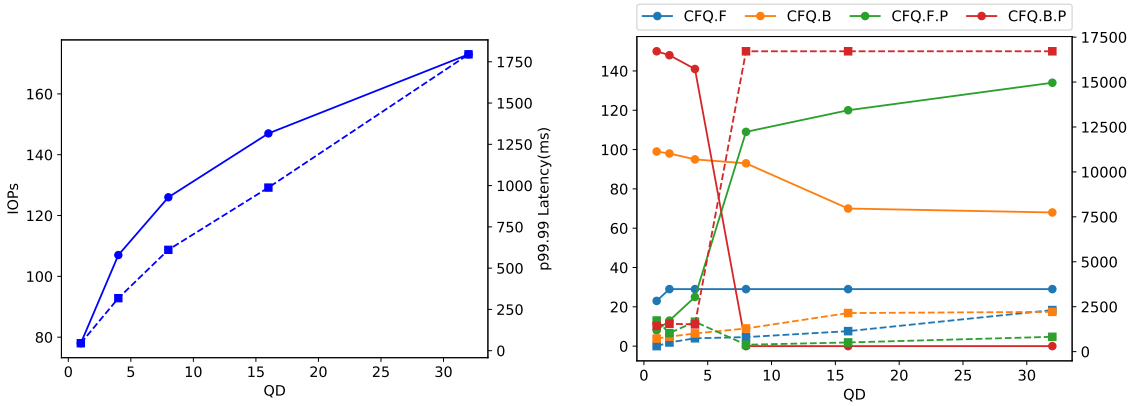
To demonstrate the performance impacts of device schedulers, we show the inversely proportional relationship of HDD throughput to IO request tail latencies by varying the number of IO requests in the device queue. It is important to understand this relationship because it is the main motivation for increasing the control over the HDD scheduler. We focus on small, random read IO commands because these commands are latency sensitive and not cacheable (writes are frequently buffered, caching avoids the storage device, large IOs have high latencies). Figure 1a captures a set of experiments where the fio tool [1] issues 4KiB random read requests to an HGST Ultrastar HE8 drive. The number of outstanding requests ranges from 1-32 and performance is measured in terms of throughput and 99.99 percentile latency. To ensure the IO requests are not served from a cache, we run fio with the DIRECT_IO option. Presented results clearly show an increase in drive throughput that correlates with the number of outstanding requests, i.e. the throughput nearly doubles as the number of outstanding requests grows from 1-32. Figure 1a also demonstrates that the increase in IOPS comes at a cost to the tail latency the application experiences. The tail latency increases from under 100ms to over a second which is a degradation of nearly 10x. With cloud storage providers pinpointing tail latency as a key metric [2, 3], we are in strong need of mechanisms that allow controlling the tail latency of individual applications.

The IO schedulers in the kernel storage stack were designed to control the latency and performance characteristics of user applications. There are three common schedulers in the Linux Kernel, CFQ, Deadline and NOOP, and only CFQ is capable of handling IO priorities. The main idea behind introducing prioritization in the host scheduler is to control the IO duration and avoid high latencies for performance critical IOs (presented in Figure 1a). In our next set of experiments we examine the effects of the CFQ prioritization on the IO request latency and total number of IOPS. We simultaneously run two types of workload on a single HDD: (i) background workload, always with 32 outstanding requests, and (ii) foreground workload with varying number of outstanding requests from 1-32. The foreground workload we run either with or without prioritization.

Figure 1b shows that CFQ achieves some level of fairness between the foreground and background workloads when priority is not used. The scheduler has the nice property that it does not starve the foreground workload in case of a low issue depth, and at the same time the background work is still able to achieve nearly half of the available IOPS. One thing to note is that the total IOPS is noticeably lower than the IOPS demonstrated in Figure 1a. In addition the CFQ scheduler does not increase foreground performance as the issue depth increases, but the background workload performance does degrade with the increased workload from the foreground. This is an unusual result because one would expect the foreground performance to increase with a decrease in IOPS for the background work. In addition, the foreground and background latencies increase with increased foreground workload. While the experiments reveal IOPS performance problems with CFQ in certain cases, the overall conclusion is that CFQ does a nice job of keeping a relatively low tail latency for foreground work. CFQ provides reasonable isolation between the foreground and background process.

In the case when IO priority is set on the foreground work, Figure 1b presents somewhat unintuitive behavior. Foreground work has a lower total IOPs and much higher latencies, when we indicate that the foreground work is high priority. In addition the background workload starts to see much higher IOPs. At low foreground queue depths we observe priority inversion. Once the queue depth of the foreground requests starts to rise to 8 requests and beyond, we see that the background work nearly stops completely and the foreground work has much better tail latencies.

In conclusion, host-level schedulers that implement priority awareness aren't very intuitive in their behavior with mixed workloads. In addition host level schedulers implementing priority leverage idling to provide isolation between priority classes, but this has the negative effect of lowering drive throughput. Due to *re-scheduling* that occurs within the drive, the host scheduler cannot make any guarantees about performance when a request is dispatched to a drive queue. To combat this problem, the schedulers idle to guarantee that all outstanding prioritized commands are finished in a predictable manner. It is our belief that our work is complementary to host-schedulers leveraging priority. We believe there is now an opportunity to revisit host-schedulers to take advantage of the benefits of device level priorities.



(a) IOPs and Latency vs QD Plot

(b) CFQ & Prio

Figure 1: **Current Behaviors of HDD and Schedulers** *The two graphs demonstrate the performance impact of increasing QD as well as how host and device schedulers can produce unexpected behavior. In all our figures IOPS are represented as solid lines and dashed lines indicate tail latency. The legend is a tuple of [Scheduler.Work.Priority On.Drive Priority], where work is F for foreground, B for background, Priority On is P, and drive priority is D. Figure 1a shows the relationship that drive queue depth has on application performance. Figure 1b demonstrates that prioritized commands and schedulers sometimes produce unexpected results.*

3 Implementation in The Linux Kernel

This section describes the prioritization extensions we introduce into the Linux Kernel. As described in Section 2, IO prioritization is a currently supported and well documented feature of the Linux kernel. However, the only IO scheduler that leverages the IO prioritization features is the CFQ scheduler. IO priorities can be assigned to a single process or a group of processes, and are manipulated via the *iopriority_set()* system call. IO requests in the Linux Kernel travel a fairly complex path from the application to the storage device. As the IO request travels from the application to the device, crossing multiple kernel layers, it is represented in several forms. To better understand the technical details of our work, and in order to describe the changes we introduce into the kernel, we further examine the IO path in the Linux Kernel and how IO prioritization fits into this path.

After being dispatched by a user application through a *read()/write()* system call, an IO request is handled by three main layers within the kernel: VFS, block and the device driver layer. Upon reaching the kernel and the VFS layer, a *bio* structure is created from the initial *read()/write()* requests, which can then be used by stackable device mapper targets. Eventually the *bio* structure is passed to the block layer where it gets transformed into a *request* structure, which is used by block layer schedulers to reorder and merge requests. CFQ is the only priority aware scheduler and obtains the priority of the IO request from the *iocontext* structure that is associated with the *task_struct* structure of the *current* running task. The *iocontext* can be manipulated with the *iopriority_set() /_get()* system calls. Based on the *iocontext* value, the *request* structure is placed in a high/low priority queue. After the scheduler dispatches a request the priority is no longer used, and the request optionally transforms into another form (typically a SCSI command) before eventually being converted by device drivers into device specific requests.

In order for IO prioritization to be a useful feature that is independent of block layer schedulers, we have made changes to the block layer of Linux. We now associate the *iocontext* priority information with the *request* when a *bio* is converted into a *request*. This allows *request* based device drivers to act on priority information, which includes SATA, SAS, and NVMe devices. In this work our focus was on enabling prioritization within SATA devices, because they are the dominant storage devices within the cloud storage stack. We have currently implemented priority support for the device in the libata layer, which is used by SATA devices and is capable of interfacing with several HBAs including AHCI (implemented in many motherboards). The reader should note that the kernel-device communication is performed through an HBA that takes requests from the kernel and internally converts them to device commands. Consequently, passing prioritized commands to a device depends on the HBA support. We also discovered that the Broadcom LSI 9300-4I4E HBA supported passing prioritized commands to SATA HDDs so we updated the driver for this HBA, mpt3sas, to support iopriorities.

4 Results

To demonstrate the IO latency effects of host→device priority communication, we repeat the experiments used to generate Figure 1b, only this time we run the experiments with priority information passed to the storage device. In addition to CFQ, we also include results generated by the NOOP and DEADLINE schedulers because our work enables IO priority to be independent of the scheduler. The results were collected on an 8 core Intel E5-2640 with 256GiB of memory running the 4.10.1 Linux kernel. All results were collected on a HGST UltraStar HE8 HDD using DIRECT_IO in order to observe storage device performance.

Figure 2 represents the performance results for multiple combinations of foreground fio workloads and available IO schedulers. As in Section 2, the background workload is fixed to 32 outstanding IO requests, while the foreground work varies this parameter from 1-32. Figure 2 shows that passing priority to the drive can improve application IOPS and tail latencies under mixed workloads. This result is consistent across all schedulers, which enables an application greater flexibility to tune their performance requirements. When we look at the results for the NOOP and DEADLINE schedulers, Figure 2a and 2b respectively, we can observe nearly identical behavior. Recall that neither of these host schedulers have priority support built into their scheduling decisions. Therefore, they do not attempt to reorder or delay requests based on the priority, instead these schedulers just pass the priority value to the drive. The internal HDD scheduler uses the passed priorities to decide about the command ordering. The results in these figures are nearly identical so we choose to examine them in tandem. In these results we see that the foreground IOPS goes up significantly for lower queue depths when prioritized requests are issued to the drive, but has diminishing returns with higher queue depths. This is expected because the drive scheduler can best perform when there is a lower ratio of high priority to default priority requests. When the ratio of prioritized commands rises then non-prioritized commands get delayed causing anti-starvation mechanisms to kick-in. This leads to the scheduler switching between requests from both the prioritized and non prioritized set of commands, reaching lower than optimal IOPS.

Figures 2a and 2b demonstrate that enabling prioritization within the device has a large positive impact on the tail latency of the foreground IO. For the foreground workload, QD 1 and NOOP scheduler, priorities reduce the tail latency from 1.4s down to 81ms, over 10x. It is interesting to note that these results are collected with the background workload fixed at 32 outstanding requests, which puts significant stress on the HDD resources. The lower tail latency is observable for all queue depths across both the DEADLINE and NOOP schedulers. For QD 1, the tail latency of the background workload increases by about 2x when priority is passed to the drive and this is also expected given the drive is forced to work on prioritized IO. Expectedly, the tail latency for the background workload further increases as the number of high priority requests is increased.

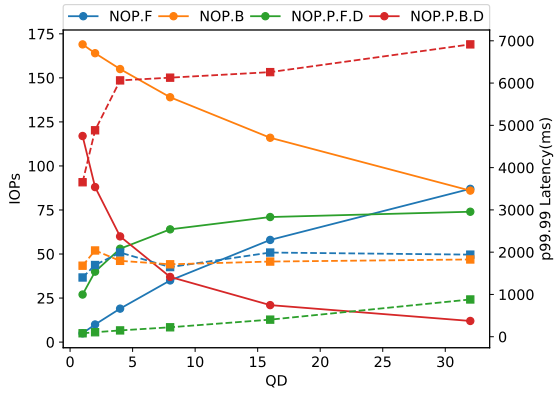
The last set of results we wish to examine are what happens to the CFQ scheduler when we pass iopriorities to the device. In Figure 2c we see that by passing iopriority to the drive, in addition to using priority in the scheduler, we are able to ensure the prioritized IO has lower latencies. Note that this figure is identical to Figure 1b with the addition of performance results of passing priority to the drive. Recall that we discussed in Section 2 that CFQ had the surprising behavior of increasing tail latency and lowering IOPS of prioritized foreground workloads, but by passing priority to the drive this is no longer an issue. We also see that IOPS are increased for foreground work with QD 4 and lower. The tail latency of the prioritized commands are vastly improved when we pass them to the device across all queue depths.

In summary by passing iopriority to the storage device, foreground IO achieves much better latency numbers and also increased IOPS in many cases. This improvement comes at a cost to the IOPS and latency of the background workload. These properties are held independently of the host level scheduler that is used.

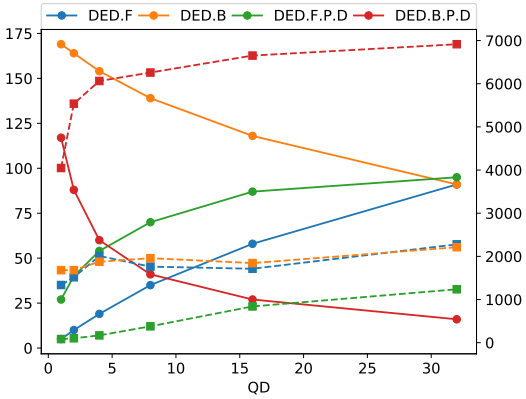
5 Related Work

IO scheduling has been the main topic of many studies in recent years. Kim et al. [12] propose connecting the IO priorities across the storage stack layers, to address the problem of priority inversions. Their approach is request-centric and based on the IO dependencies. Yang et al. introduce split scheduling [18], a handler-based cross-layer scheduling that enables preserving the IO information. Split scheduling prevents host-side storage layers, from reordering an IO request. IOFlow [17] provides IO differentiation information between storage hypervisors and servers. Our method can be seen as complementary to these approaches, because we extend prioritized host based-scheduling to a device.

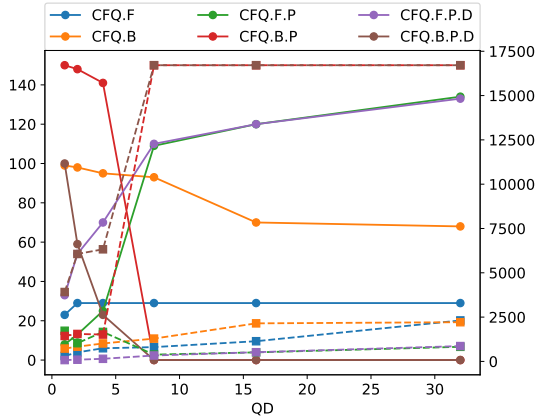
Young Jin et al. [19] address the issue of host-IO Device re-scheduling. They acknowledge the host-device IO *re-scheduling* and design a dynamic scheduler that allows switching on/off the host and the device schedulers at runtime. They do not attempt passing the IO priorities to the device. A set of real-time scheduling algorithms has been developed [16, 8, 16]. While they rely on SCAN to deliver highest throughput, we allow our priority commands to be processed completely independently of the standard drive request processing algorithm. Other traditional schedulers [11, 14, 13] allow IO priorities, but



(a) NOOP Scheduler



(b) DEADLINE Scheduler



(c) CFQ Scheduler

Figure 2: **Drive Priority & Scheduler Impact** *These graphs demonstrate that priority information passed to the storage device improves latency across all host level schedulers.*

do not pass them to the IO device. Differentiated Storage [15] proposed modifications to IO interfaces to provide application to device IO classification, whereas we reuse existing interfaces and infrastructure.

Multiple studies [9, 5, 10] investigate IO scheduling in distributed systems, by mixing IO- and computation--bound workloads on a single node and focusing on the effects of multi-level IO scheduling in a virtualized/cloud environment. These studies are limited to the host side and do not include priority-aware IO devices. Blagojević et al. [6] show the importance of IO scheduling in a cloud environment and address the device interaction with a distributed IO accesses. In our work we systematically examine the Linux kernel changes necessary to allow IO prioritization within the device. Our work allows transparent integration of priorities with distributed systems and local host software.

6 Conclusion & Future Directions

In this work we have shown that using iopriority within a storage device improves prioritized application tail latencies and IOPS. Our approach is complementary to existing priority handling in host level schedulers and we have implemented our changes in the Linux Kernel and these changes have been merged upstream. The results show that prioritized commands to the device improve foreground performance across three Linux schedulers. This work is a step in the direction of allowing finer grained control of storage device behavior. Although we have demonstrated our work on an HDD it is applicable to SSDs as well, because they also queue and delay requests internally. With cloud storage providers being heavily dependent on predictable tail latencies this work provides a new dimension of optimization which is critical for cost and performance sensitive applications.

Our future work will focus on examining more workloads and including mixed read/write workloads. Prioritization is currently implemented at the process level and we plan to investigate command level prioritization with aio interfaces in the Linux Kernel. In addition we will look at the internals of host level schedulers and identify areas where knowledge of device priority may lead to alternative design choices. This work has focused on HDD performance and another direction of the future work will be an SSD extension. In addition, the NVMe standards have command and queue level prioritization and we plan to investigate ioprioritization in this context.

References

[1] *https://github.com/axboe/fio*.

[2] *https://research.googleblog.com/2009/06/speed-matters.html*.

[3] *http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf*.

[4] Linux 4.10 kernel. Accessed: 2017-03-09.

[5] ARAGIORGIS, D., NANOS, A., AND KOZIRIS, N. Coexisting scheduling policies boosting i/o virtual machines. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2* (2012), Euro-Par'11, Springer-Verlag, pp. 407–415.

[6] BLAGOJEVIC, F., GUYOT, C., WANG, Q., TSAI, T., MATEESCU, R., AND BANDIC, Z. Priority IO scheduling in the cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'13, San Jose, CA, USA, June 25-26, 2013* (2013).

[7] BREWER, E., YING, L., GREENFIELD, L., CYPHER, R., AND T'SO, T. Disks for data centers. Tech. rep., Google, 2016.

[8] CHANG, H.-P., CHANG, R.-I., SHIH, W.-K., AND CHANG, R.-C. Reschedulable-group-scan scheme for mixed real-time/non-real-time disk scheduling in a multimedia system. *J. Syst. Softw. 59*, 2 (Nov. 2001), 143–152.

[9] HU, Y., LONG, X., ZHANG, J., HE, J., AND XIA, L. I/o scheduling model of virtual machine based on multi-core dynamic partitioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (New York, NY, USA, 2010), HPDC '10, ACM, pp. 142–154.

[10] IBRAHIM, S., JIN, H., LU, L., HE, B., AND WU, S. Adaptive disk i/o scheduling for mapreduce in virtualized environment. In *Proceedings of the 2011 International Conference on Parallel Processing* (Washington, DC, USA, 2011), ICPP '11, IEEE Computer Society, pp. 335–344.

[11] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 117–130.

[12] KIM, S., KIM, H., LEE, J., AND JEONG, J. Enlightening the i/o path: A holistic approach for application performance. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 345–358.

[13] LUMB, C. R., SCHINDLER, J., AND GANGER, G. R. Freeblock scheduling outside of disk firmware. In *Proceedings of the 1st USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2002), FAST'02, USENIX Association, pp. 20–20.

[14] LUMB, C. R., SCHINDLER, J., GANGER, G. R., NAGLE, D. F., AND RIEDEL, E. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4* (Berkeley, CA, USA, 2000), OSDI'00, USENIX Association, pp. 7–7.

[15] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 57–70.

[16] REDDY, A. L. N., WYLLIE, J., AND WIJAYARATNE, K. B. R. Disk scheduling in a multimedia i/o system. *ACM Trans. Multimedia Comput. Commun. Appl. 1*, 1 (Feb. 2005), 37–59.

[17] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 182–196.

[18] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 474–489.

[19] YU, Y. J., SHIN, D. I., EOM, H., AND YEOM, H. Y. Ncq vs. i/o scheduler: Preventing unexpected misbehaviors. *Trans. Storage 6*, 1 (Apr. 2010), 2:1–2:37.