# File Fragmentation from the Perspective of I/O Control

Jonggyu Park
Sungkyunkwan University
jonggyu@skku.edu

Young Ik Eom
Dept. of Electrical and Computer Engineering /
College of Computing and Informatics,
Sungkyunkwan University
yieom@skku.edu

## ABSTRACT

File fragmentation has been widely studied for several decades due to its detrimental effects on I/O activities. However, most of the previous research focuses on its performance aspect in a single application. In this paper, we analyze the effect of fragmentation on I/O control in a consolidated system where multiple applications run simultaneously. Our evaluation demonstrates that all of the weight-based I/O control mechanisms supported by the Linux kernel fail to achieve fair I/O sharing for different reasons when they meet fragmentation. Also, we show that defragmentation can promptly antidote such failures by preventing request splitting and device-level resource conflicts.

## CCS CONCEPTS

• **Information systems** → *Storage management*; • **Software and its engineering** → *Software maintenance tools.*

## KEYWORDS

File fragmentation, I/O control, cgroups

**ACM Reference Format:**
Jonggyu Park and Young Ik Eom. 2022. File Fragmentation from the Perspective of I/O Control. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22), June 27–28, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3538643.3539746

## 1 INTRODUCTION

Resource management has been a building block for realizing application consolidation [15, 17, 23–25, 28, 30, 32, 34, 35, 38]. Since each application has a discrete performance requirement, the underlying system should provide an adequate amount of system resources. Additionally, to guarantee a certain level of performance regardless of the behavior of co-running applications, the system should prevent performance interference among applications, thereby providing strict performance isolation. Modern Linux systems utilize Cgroup [5, 6] to fulfill such necessities. Especially, the weight-based I/O controlling of Cgroup is widely utilized due to its convenience, intuitive design, and work conservation [16].

To control I/O resources, Cgroup tightly communicates with I/O controllers at the block layer, such as CFQ (Completely Fair Queuing) [3] and BFQ (Budget Fair Queuing) [1]. Specifically, each I/O controller refers to the I/O weight values of applications and controls/limits their I/O requests according to the I/O weights. However, we observe that file fragmentation hinders I/O controller from precisely controlling I/Os for various reasons. Unfortunately, previous studies about fragmentation [8, 12, 14, 19, 21, 22, 26, 29, 31, 33] only focus on the I/O performance of a single application without exploring performance interference among applications. Therefore, its negative influence on I/O control remains undiscovered.

In this paper, we approach filesystem fragmentation from the perspective of I/O control and discover various problems incurred by fragmentation in a consolidated system where multiple applications run together. We first investigate the internal behavior of I/O control mechanisms that support Cgroup and discover the cases of scheduling failures incurred by fragmentation. Afterward, we eliminate existing fragmentation using FragPicker [27], a defragmenter optimized for modern SSDs, and analyze how defragmentation affects I/O control in such a system.

In summary, our results demonstrate that fragmentation increases the number of I/Os, decreases the size of each I/O, and complicates the performance characteristics. Due to such overheads, fragmentation dilutes the effectiveness of the conventional I/O control mechanisms. First, CFQ experiences a failure of I/O control because its IOPS-based I/O control does not consider the size of each I/O operation. Second, BFQ incurs performance interference since its sector-based I/O control sacrifices certain applications to equalize I/O bandwidth across applications. Finally, the linear performance


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

ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539746

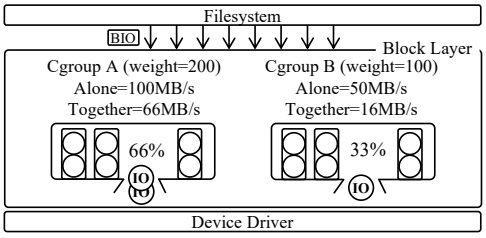


**Figure 1: An Example of Proportional I/O Sharing**

model of IOCost [16] fails to predict the I/O performance and thus incurs unfair I/O scheduling. We observe that all of the problems can be instantly mitigated by defragmentation.

# 2 BACKGROUND

## 2.1 Cgroup and I/O Control

Various Linux systems utilize Cgroup for controlling system resources, and its importance has been increasing due to the widespread adoption of application consolidation. For example, contemporary server systems consolidate various server instances using virtualization technology for better resource efficiency [9, 36, 37]. In this circumstance, Cgroup plays a critical role in resource management to guarantee the performance requirement of each server while minimizing performance interference. Similarly, mobile systems also utilize Cgroup to boost the performance of the foreground application for better user responsiveness while preventing resource monopolization by background applications [4].

There are various ways to regulate I/O resources. First, blk-throttle [2] limits I/Os in the form of available read/write IOPS or bandwidths. However, this method is not work-conserving. Second, IOLatency [7] can set a target I/O latency per cgroup. Unfortunately, finding an appropriate I/O latency in the real world is arduous [16]. Finally, one can set I/O weight of each cgroup and proportionally distribute I/O resources according to their I/O weights.

For example, as shown in Figure 1, suppose cgroup A has an I/O weight of 200 and cgroup B has an I/O weight of 100. In this case, the I/O control mechanism in the block layer regulates the I/O activities of each cgroup and provides 66% and 33% of I/O resources to cgroup A and B, respectively. Therefore, if cgroup A can achieve 100MB/s when it runs alone, its bandwidth should be at least 66MB/s regardless of I/O behaviors of other cgroups. Similarly, the system should guarantee at least 16MB/s of throughput to cgroup B, which is around 33% of 50MB/s. To provide performance isolation, the application behavior in a single cgroup should not subvert such promised performance guarantees of all the cgroups.

*2.1.1 Completely Fair Queuing.* To realize such proportional I/O sharing of Cgroup, the Linux kernel supports various I/O control mechanisms. First, CFQ [3] is an I/O scheduler for the single-queue block layer that distributes I/O resources based on timeslice or IOPS. CFQ maintains per-process I/O queues and dispatches I/Os from the I/O queues by rotating the I/O queues whenever each I/O queue expires its timeslice. To support group scheduling, CFQ manages a cfq group for each cgroup and keeps track of their I/O usages using per-cgroup vdisktime. Afterwards, CFQ picks a cgroup that has the smallest vdisktime and serves their I/Os. Since it is not feasible to accurately measure the elapsed time in the case of NCQ (Native Command Queuing)-enabled devices, CFQ utilizes IOPS instead of timeslice in calculating vdisktime, if the device is non-rotational and NCQ-enabled.

However, CFQ is vulnerable to the case where heterogeneous workloads co-run. For example, if a high-priority application issues small-sized I/Os while a low-priority one issues large-sized I/Os, the low priority one can monopolize the storage devices since CFQ utilizes IOPS without considering the actual size of each I/O.

*2.1.2 Budget Fair Queuing.* Second, BFQ [1] is a multi-queue I/O scheduler that distributes a fair amount of I/O bandwidth to each application. Unlike CFQ, BFQ utilizes the aggregate amount of I/O sectors issued by each application, instead of either elapsed time or IOPS. Therefore, BFQ can achieve fairness in terms of I/O bandwidths among applications. However, BFQ can incur performance interference since it equalizes the I/O bandwidth without considering the influence of each I/O on the storage. For example, it has been reported that larger-sized I/Os can achieve a higher I/O performance [10, 13, 18, 20]. Therefore, BFQ allows an application issuing small-sized I/Os to occupy the underlying devices longer than another issuing large-sized I/Os, in order to equalize their I/O bandwidths. In this way, with BFQ, the performance of applications can be significantly affected by the I/O characteristics of other applications.

*2.1.3 IOCost.* Finally, a new I/O control mechanism, IOCost [16], has been proposed to overcome the drawbacks of the previous I/O schedulers. It predicts the cost of each I/O using a linear performance model and calculates the available bandwidth for each application. In this way, IOCost can achieve a fair amount of device occupancy for each application. However, the linear performance model is insufficient to predict the performance of SSDs because the I/O performance varies depending on the location of the data inside the storage even when their I/O attributes (I/O size, randomness, etc.) are identical.

## 2.2 Fragmentation and Its Negative Influence

File fragmentation refers to the state that data are scattered into multiple non-contiguous pieces instead of a single large

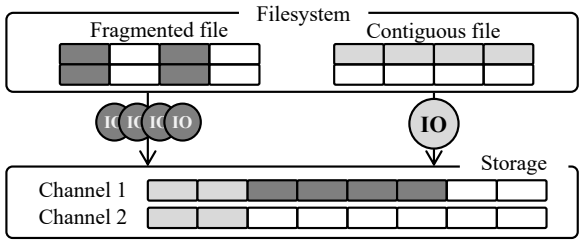


**Figure 2: File Fragmentation and its Influence**

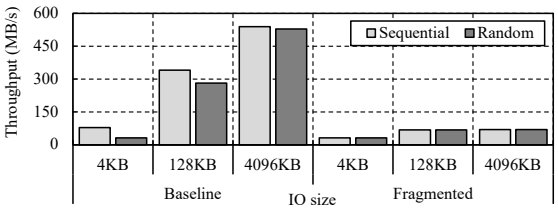


**Figure 3: The Performance Overheads Induced by Fragmentation**

one. It has been continuously reported that fragmentation occurs regardless of the filesystem type and degrades the I/O performance [11, 21, 27]. As shown in Figure 2. Fragmentation at the filesystem layer incurs request splitting where a single I/O request is split into multiple ones. This reduces the average I/O size and increases both the number of I/Os and their randomness for the same amount of data. Fragmentation at the device layer hinders the exploitation of the parallel units inside SSDs by incurring channel conflicts.

To explore the performance degradation by fragmentation, we generate a fragmented file with 4KB fragment size and measure the performance of sequential/random read operations with various I/O sizes. Since the file is fragmented into 4KB fragments, all the I/Os are eventually split into 4KB-sized I/O requests/commands regardless of the size of system calls that the application issues. Here, baseline shows the performance of non-fragmented files. All experiments in this paper are performed on a machine with Intel E5-2620 v4 CPU, 128 GiB RAM, and Samsung Flash SSD 850 Pro 256GB. We use the F2FS filesystem with Linux kernel 5.7.0 (4.19.176 for CFQ).

As shown in Figure 3, the performance of sequentially reading fragmented files is similar to that of randomly reading non-fragmented files with 4KB I/Os. Specifically, fragmentation decreases the I/O performance by around 80% when issuing 128KB sequential reads. This results comes from the fact that fragmentation increases the randomness of I/Os while decreasing the average size of each I/O.

Since SSDs utilize internal parallelism to improve I/O throughput [10, 13, 18, 20, 39], the data placement is highly

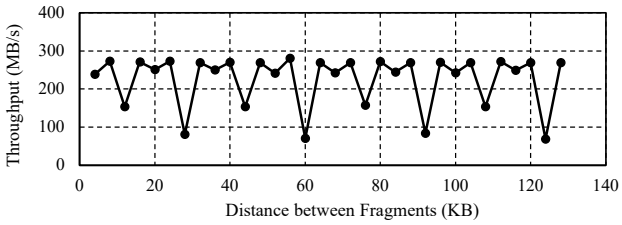


**Figure 4: The Sequential Read Performance with Varying Distance btw Each Fragment**

correlated to effectiveness of device-internal resources, thereby influencing the I/O performance. To experimentally demonstrate this phenomenon, we generate a fragmented file with 4KB fragments and vary the distance between each fragment. To generate both filesystem-level and device-level fragmentation in the desired way, we utilize the attribute of the log-structured allocation of F2FS and the Flash SSD, which allocates new slots in the arrival order. Specifically, using O_DIRECT I/Os, we append 4KB of data to the target file and then append dummy data to a dummy file. Here, we set the size of the dummy data for appending as the specified distance. This process is serialized so that all of the write operations are delivered to the storage without re-ordering. By repetitively performing these operations, the target file can be stored in an interleaved way with the dummy file in both the filesystem and the storage.

In this experiment, all the aspects including the average I/O size and the number of I/Os at the block layer are identical except for the distance between fragments. Figure 4 shows the 128KB sequential read performance. The flash SSD shows various performance results depending on the distance. Specifically, it experiences a significant performance drop at every 32KB (4KB * 8) distance by showing up to 70.5% lower performance than other cases. This indicates that the SSD consists of multiple parallel units that can handle 32KB at a time, and their conflicts significantly degrade the performance. Like this, the data placement determines the I/O performance even when all the other I/O characteristics are identical. This anomaly cannot be detected by the linear performance model of IOCost, thereby obstructing its I/O control.

## 3 FAILURE OF I/O CONTROL DUE TO FRAGMENTATION

File fragmentation decreases the average I/O size and increases both the number of I/Os and their randomness. We observe that such overheads incurred by fragmentation exacerbate the drawbacks of each I/O control mechanism. To experimentally demonstrate it, we perform the following

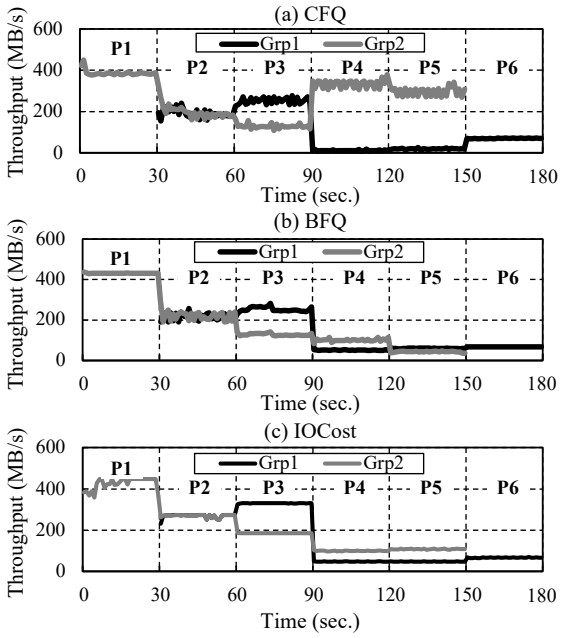


**Figure 5: Scheduling Failure of I/O Control Mechanisms due to Fragmentation**

experiments. First, [P1] we run a workload that reads a non-fragmented file with 128KB O_DIRECT I/Os in group 2 (grp 2). Second, [P2] after 30 seconds, we run the same workload in group 1 (grp1) to monitor if each I/O control mechanism can properly manage I/O resources. Third, [P3] we increase the I/O weight of grp1 from 100 to 200 so that the weight of grp1 becomes double the weight of grp2. Fourth, [P4] we restore the priority of grp1 to 100 and switch the target file of grp1 to a fragmented file. Here, the size of each fragment is 4KB, and the distance between two fragments is 28KB. In this way, the first offset of a fragment is 32KB away from that of the next fragment. Afterward, [P5] we again increase the weight of grp1 to 200. Finally, [P6] we terminate grp2 so that grp1 can run alone. We configure the experiment in this way to compact various possible situations into a single figure.

Using this benchmark, we evaluate the effectiveness of CFQ, BFQ, and IOCost when they meet fragmentation. As shown in Figure 5, all the I/O control mechanisms achieve their scheduling goal when both grp1 and grp2 read non-fragmented files (P2 and P3). Specifically, in P2, grp1 and grp2 show identical performance since they have identical I/O weights and run the same workload. In P3, grp1 achieves two times higher throughput than grp2, which conforms to the I/O weights. However, all of them fail to achieve their goals when they meet fragmentation.

To provide performance isolation, the behavior of one cgroup should not influence the performance of other cgroups, which means the performance of grp2 in P4 and P5 should be the same as that in P2 and P3, respectively. Additionally, to provide proportional I/O sharing, the performance of grp1 and grp2 in P4 should be at least 50% of that in P6 (grp1 alone) and in P1 (grp2 alone), respectively, since they should equally share the I/O resources (50%:50%) in P4. In other words, each of the cgroups in P4 should show at least half of the performance when each of them runs alone.

However, CFQ fails to achieve proportional I/O sharing due to its IOPS-based policy. At P4, the performance of grp1 is only 15.5% of the performance when it runs alone (P6). Since grp1 at P4 and P5 reads fragmented files, it requires a more number of I/Os for the same amount of data than grp2. Specifically, reading fragmented files (grp1) requires around 32 times more number of I/Os for the same amount of data, compared with contiguous files (grp2). Therefore, the IOPS-based policy of CFQ allows grp2 to occupy a larger amount of I/O resources in order to equalize the number of I/Os across cgroups at P4.

In the case of BFQ, since BFQ tries to fairly distribute I/O bandwidths across applications, fragmentation incurs performance interference. To achieve its scheduling goal, BFQ provides more I/O resources to grp1 because grp1 needs more I/O resources to process the same amount of I/O sectors, compared with grp2. In more detail, the average I/O size of grp1 at P4 is only 4KB while that of grp2 is around 128KB. Meanwhile, due to the smaller I/O size, grp1 requires around 5 times longer I/O time in processing the designated amount of I/O sectors, compared with grp2. Therefore, grp2 experiences performance degradation when grp1 reads fragmented files. Particularly, the throughput of grp2 in P5 is around 70% lower than that in P3.

IOCost calculates the device occupancy of each I/O operation using the linear performance model. Unfortunately, the Flash SSD shows considerably different performance results depending on the data layout even when the I/O characteristics at the block layer are identical. Fragmentation scrambles the data placement, thereby decreasing the accuracy of the performance model. Although IOCost has a dynamic adjustment module to handle unexpected I/O performance, it is insufficient to deal with the performance anomaly of fragmentation. Therefore, IOCost also suffers from performance interference. Specifically, grp2 experiences a 63% performance drop due to the workload change of grp1, when comparing P4 with P2 in Figure 5.

Like this, fragmentation brings about scheduling failure of each I/O control for different reasons. First, an increase in the number of I/Os due to fragmentation obstructs the scheduling of CFQ. Second, BFQ experiences performance interference due to an increased amount of I/O resources required for the same amount of data. Finally, fragmentation decreases the accuracy of the performance model of IOCost, thereby incurring the failure of effective I/O control.

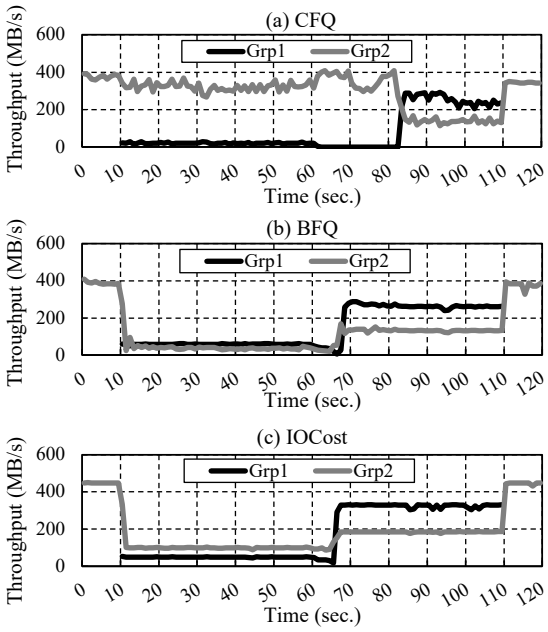


**Figure 6: Defragmentation Effect on I/O Control with Homogeneous Workloads**

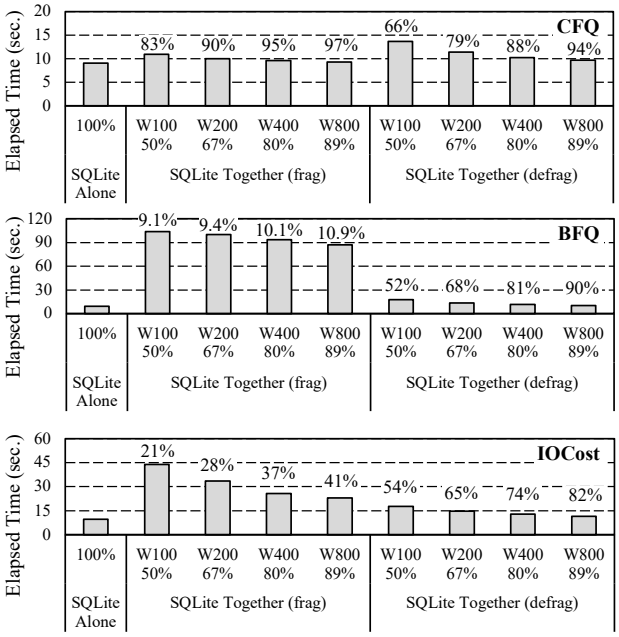


**Figure 7: Elapsed Time of SQLite SELECT Query while Varying I/O Weight**

# 4 DEFRAGMENTATION AS A REMEDY OF SCHEDULING FAILURE

In this section, we remedy the cases of scheduling failures using defragmentation which eliminates existing fragmentation by relocating filesystem blocks. For defragmentation, we utilize FragPicker [27].

## 4.1 Homogeneous Workloads

To accurately investigate the effect of defragmentation on I/O control, we run a workload that performs 128KB sequential I/Os with O_DIRECT in two different cgroups, where grp1 has a two times higher I/O weight than grp2. Therefore, grp1 should occupy 2/3 of the entire I/O resources while grp2 takes 1/3. Here, grp1 reads fragmented files whereas grp2 reads non-fragmented files. We run grp2 first with running grp1 after 10 seconds. Afterward, we perform FragPicker at around Time 60. Here, we run FragPicker inside grp1 so that the I/Os from FragPicker belong to grp1, because FragPicker defragments only the files of grp1. At Time 110, we suspend grp1 to measure the performance of grp2 when it runs alone.

Figure 6 shows the performance of both grp1 and grp2 in a time-series manner. In the case of CFQ, as grp1 begins its task at Time 10, grp1 suffers from a significant performance drop because the IOPS-oriented policy of CFQ allows grp2 to monopolize the storage devices. Specifically, grp1 running with grp2 shows only 28% of the throughput, compared with grp1 running alone. After defragmentation, CFQ achieves fairer I/O sharing, thereby grp1 achieving 1.79 times higher throughput than grp2. Here, since the performance of FragPicker is also degraded by grp2 due to the same reason, FragPicker with CFQ requires a much longer time for defragmentation than that with BFQ and IOCost.

With BFQ and IOCost, grp2 suffers from significant performance interference by grp1 before defragmentation. After defragmentation is performed, both BFQ and IOCost properly control the I/O resources according to the I/O weights by showing around two times higher throughput for grp1, compared with grp2.

## 4.2 Heterogeneous Workloads

To investigate the case of heterogeneous workloads, we perform SELECT query to SQLite database with one million entries (4.4GB in total) while running 64KB random read with libaio (iodepth:16) using FIO benchmark, in two different cgroups. Also, we vary the I/O weight of the SQLite workload from 100 to 800. To measure the effect of fragmentation, we perform the evaluation two times, one (frag) with FIO reading fragmented files and another (defrag) after defragmenting the FIO files. In the figures, the percentage on x-axis denotes its I/O share, and one on each bar means the relative performance to the case when the workload runs alone. Therefore, the percentage value on the x-axis and the bar should be matched to provide proportional I/O sharing.

Figure 7 shows the elapsed time of SELECT query. Similarly to the previous evaluations, BFQ and IOCost cannot

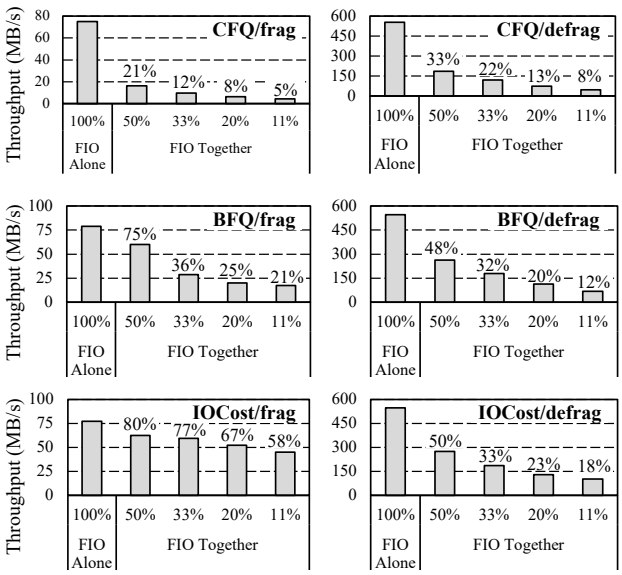


**Figure 8: Throughput of FIO Benchmark before/after Defragmentation**

provide performance isolation when FIO reads fragmented files. Therefore, the elapsed time of SELECT query significantly increases due to FIO. In the meantime, CFQ provides more I/O resources to SQLite workload than its share since its average I/O size is bigger than FIO with fragmented files. After defragmentation, all the I/O control mechanisms show better I/O proportionality.

Figure 8 presents the throughput of the FIO benchmark. Before defragmentation, CFQ provides less I/O resources to the FIO workload than its share whereas BFQ and IOCost provides more I/O resources than its share. However, defragmentation can mitigate the unfair I/O sharing of I/O control mechanisms by eliminating request splitting and minimizing resource conflicts inside the storage device. In particular, BFQ achieves nearly ideal I/O proportionality after defragmentation.

## 5 DISCUSSION

As shown in the paper, defragmentation can promptly relieve the scheduling failures caused by fragmentation. However, since defragmentation generates additional write operations, it incurs several problems in practice. First, the additional write operations can curtail the device lifespan. Second, defragmentation can degrade the co-running applications, which is detrimental to consolidated environments. To mitigate these overheads, several studies have been proposed in recent years. For example, to minimize the amount of writes, FragPicker [27] migrates only a certain portion of data that can benefit from defragmentation. Janusd [14] is a copyless defragmentor, which utilize a device-level remap operation to migrate data.

In addition to the previous efforts for minimizing the amount of writes, we argue that defragmentation should be applied in a different manner, depending on I/O control mechanisms. This paper observed that fragmentation with CFQ degrades the performance of applications that are actually accessing the fragmented files. On the other hand, fragmentation with BFQ/IOCost degrades even the performance of other unrelated applications. Therefore, to improve the performance of high-priority applications with BFQ and IOCost, we should check the fragmentation states of other applications as well as the high-priority ones. Also, to minimize the performance interference of defragmentation with high-priority applications, it can be a possible method to have the defragmentation process in its own cgroup and regulate its I/O rate, or attribute the I/Os for data migration to the owner cgroup as with the experiments in this paper.

Finally, analyzing proper data area for migration can decrease the amount of writes for defragmentation. As Park *et al.* [27] argued, request splitting can be a primary factor in performance degradation incurred by fragmentation on some SSDs. In such cases, migrating only certain fragmented data that causes request splitting can reduce the amount of writes for defragmentation. Similarly, since the frequency of data accesses is not always uniform in practice, not all data is equally critical to the application performance. Therefore, we believe that identifying hot data (frequently accessed) for migration either at the system call layer [27] or at the block layer can efficiently minimize the amount of writes for defragmentation.

## 6 CONCLUSION

In this paper, we analyze file fragmentation from the perspective of I/O control. We discover that various I/O control mechanisms fail to achieve their goals for different reasons, when they meet fragmentation. Also, we prove that defragmentation can be one of the options for promptly solving such problems. We believe the experimental results in this paper will be helpful to those who design a new I/O control mechanism or address fragmentation-related issues in consolidated environments.

## ACKNOWLEDGMENTS

We thank our shepherd, Joo-young Hwang, and the anonymous HotStorage reviewers for their invaluable feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays).

# REFERENCES


[1] BFQ (Budget Fair Queueing). https://www.kernel.org/doc/html/latest/block/bfq-iosched.html.

[2] block-throttle: proportional throttle. https://lwn.net/Articles/676823/.

[3] CFQ (Complete Fair Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[4] Cgroup Abstraction Layer. https://source.android.com/devices/tech/perf/cgroups.

[5] Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[6] Cgroups v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[7] Introduce io.latency io controller for cgroups. https://lwn.net/Articles/758697/.

[8] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2009. Generating realistic impressions for file-system benchmarking. In *Proc. USENIX FAST*. 125–138.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Operat. Syst. Rev.* 37, 5 (2003), 164–177.

[10] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. ACM SIGMETRICS*. 181—-192.

[11] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. 2017. File systems fated for senescence? nonsense, says science!. In *Proc. USENIX FAST*. 45–58.

[12] Giel de Nijs, Ard Biesheuvel, Ad Denissen, and Niek Lambert. 2006. The effects of filesystem fragmentation. In *Proc. OLS*. 193–208.

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

[14] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. 2017. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proc. USENIX ATC*. 759–771.

[15] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. 2019. Multi-queue fair queuing. In *Proc. USENIX ATC*. 301–314.

[16] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. 2022. IOCost: Block io control for containers in datacenters. In *Proc. ACM ASPLOS*. 595–608.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *Proc. USENIX NSDI*. 295–308.

[18] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. 2013. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Trans. on Comput.* 62, 6 (2013), 1141–1155.

[19] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. 2018. File fragmentation in mobile devices: measurement, evaluation, and treatment. *IEEE Trans. on Mobile Computing* 18, 9 (2018), 2062–2076.

[20] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting widely held ssd expectations and rethinking system-level implications. *SIGMETRICS Perform. Eval. Rev.* 41, 1 (2013), 203–216.

[21] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. 2018. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *Proc. USENIX ATC*. 691–703.

[22] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. 2020. Countering fragmentation in an enterprise storage system. *ACM Trans. Storage* 15, 4 (2020), 1–35.

[23] J. Kim, E. Lee, and S. H. Noh. 2016. I/O scheduling schemes for better i/o proportionality on flash-based ssds. In *Proc. IEEE MASCOTS*. 221–230.

[24] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. 2020. Dc-store: Eliminating noisy neighbor containers using deterministic i/o performance and resource isolation. In *Proc. USENIX FAST*. 183–191.

[25] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2016. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proc. ACM SIGCOMM*. 144–159.

[26] Jonggyu Park and Young Ik Eom. 2020. Anti-aging lfs: Self-defragmentation with fragmentation-aware cleaning. *IEEE ACCESS* 8 (2020), 151474–151486.

[27] Jonggyu Park and Young Ik Eom. 2021. Fragpicker: A new defragmentation tool for modern storage devices. In *Proc. ACM SOSP*. 280–294.

[28] Jonggyu Park and Young Ik Eom. 2021. Weight-aware cache for application-level proportional i/o sharing. *IEEE Trans. Comput.* (2021), 1–14.

[29] Jonggyu Park, Dong Hyun Kang, and Young Ik Eom. 2016. File defragmentation scheme for a log-structured file system. In *Proc. ACM APSys*. 1–7.

[30] Jonggyu Park, Kwonje Oh, and Young Ik Eom. 2020. Towards application-level I/O proportionality with a weight-aware page cache management. In *Proc. IEEE MSST*. 1–11.

[31] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. 1995. File system logging versus clustering: A performance comparison. In *Proc. USENIX ATC*. 1–21.

[32] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. USENIX OSDI*. 349–362.

[33] Keith A. Smith and Margo I. Seltzer. 1997. File system aging—increasing the relevance of file system benchmarks. In *Proc. ACM SIGMETRICS*. 203–213.

[34] Shanjiang Tang, Bu-Sung Lee, and Bingsheng He. 2016. Fair resource allocation for data-intensive computing in the cloud. *IEEE Transactions on Services Computing* 11, 1 (2016), 20–33.

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

[36] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. 2005. Intel virtualization technology. *IEEE Comput.* 38, 5 (2005), 48–56.

[37] Werner Vogels. 2008. Beyond Server Consolidation: Server consolidation helps companies improve resource utilization, but virtualization can help in other ways, too. *ACM Queue* 6, 1 (2008), 20–26.

[38] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. 2021. D2FQ: Device-direct fair queueing for nvme ssds. In *Proc. USENIX FAST*. 403–415.

[39] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an unwritten contract of intel optane SSD. In *Proc. USENIX HotStorage*. 1–8.