NCQ-Aware IO Scheduling for Conventional Solid State Drives-revise

Conference Paper · May 2019

CITATIONS
0

READS
108

4 authors, including:

Song Wu
Huazhong University of Science and Technology
190 PUBLICATIONS **1,626** CITATIONS
SEE PROFILE

Shadi Ibrahim
National Institute for Research in Computer Science and Control
56 PUBLICATIONS **946** CITATIONS
SEE PROFILE

Hai Jin
Huazhong University of Science and Technology
901 PUBLICATIONS **6,784** CITATIONS
SEE PROFILE

NCQ-Aware I/O Scheduling for Conventional Solid State Drives

Hao Fan*, Song Wu*, Shadi Ibrahim†, Ximing Chen*, Hai Jin*, Jiang Xiao* and Haibing Guan‡
* National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, China
† Inria, IMT Atlantique, LS2N, France
‡ Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China
Email: {fanh, wusong, xmchen, hjin, jiangxiao}@hust.edu.cn, shadi.ibrahim@inria.fr, hbguan@sjtu.edu.cn

***Abstract*—While current fairness-driven I/O schedulers are successful in allocating equal time/resource share to concurrent workloads, they ignore the I/O request queueing or reordering in storage device layer, such as *Native Command Queueing* (NCQ). As a result, requests of different workloads cannot have an equal chance to enter NCQ (NCQ conflict) and fairness is violated. We address this issue by providing the first systematic empirical analysis on how NCQ affects I/O fairness and SSD utilization and accordingly proposing a NCQ-aware I/O scheduling scheme, NASS. The basic idea of *NASS* is to elaborately control the request dispatch of workloads to relieve NCQ conflict and improve NCQ utilization. *NASS* builds on two core components: an evaluation model to quantify important features of the workload, and a dispatch control algorithm to set the appropriate request dispatch of running workloads. We integrate *NASS* into four state-of-the-art I/O schedulers and evaluate its effectiveness using widely used benchmarks and real world applications. Results show that with NASS, I/O schedulers can achieve 11-23% better fairness and at the same time improve device utilization by 9-29%.**

***Index Terms*—SSD, NCQ, I/O scheduler, concurrent I/O**

I. Introduction

Solid State Device (SSD) is widely deployed for critical data-intensive applications [1]. This is due to its high access performance and decreasing price [2]. A SSD is usually constructed with several channels. Each channel contains a number of chips. This design provides rich parallelism which result is high I/O performance, but often introduces important I/O interferences among workloads sharing the SSD [1]. This poses a significant challenge when offering I/O fairness between concurrent workloads (e.g., concurrent applications in an operating system, concurrent *virtual machines* (VMs) on a shared host) while ensuring high device utilization.

There are two primary ways to address this challenge: (i) Relying on device customization at hardware layer (e.g., *Flash Translation Layer* (FTL) or Open Channel) [1, 3–9], however, these solutions require special hardware supports and thus are hard to be applied to conventional SATA-based SSDs, which still dominate SSD market [10]. (ii) Relying on SSD-friendly I/O schedulers [11–15] which leverages SSD features (e.g. read/write asymmetric [11, 12] and garbage collection (GC) [13]). While these schedulers are in large part successful, they mostly ignore I/O request queueing which is an important layer in SATA-based SSDs.

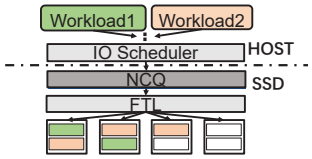


Fig. 1: NCQ is an important layer existing between I/O scheduler and FTL

I/O request queueing, such as *native command queueing* (NCQ) [16] in SATA and *submission queue* (SQ) [9] in NVMe, can be considered as the junction of operating system and storage device. They are adopted to fully exploit parallelism in SSDs. Taking widely used SATA-based SSD for example, NCQ receives requests from I/O schedulers and dispatches them concurrently to FTL on SSD, as shown in Figure 1. Therefore, fairness can be only achieved if requests of different workloads have equal chances to enter SSD. Hence, we argue that current I/O schedulers [11, 12, 14, 15] may fail in practice to achieve the desired fairness and device utilization without taking NCQ into consideration.

Accordingly, a series of experiments have been conducted to evaluate how NCQ affects I/O fairness and SSD utilization. Our experiments have revealed that NCQ conflict (i.e., when requests of different workloads cannot have an equal chance to enter SSD) occurs when aggressive workload[1] occupies most of the NCQ when it runs concurrently with a non-aggressive workload — this is common in the cloud, for example, VMs holding aggressive workloads (e.g. video, Hadoop, and mysql), which have relatively high request arriving speed and big request size, run together with VMs holding non-aggressive workloads (e.g. web and mail) on one host; and as a result of sequentiality-driven optimization [17] (i.e., I/O merging). This, in turn, harms the fairness of I/O schedulers. Moreover, we observe that NCQ utilization, which indicates the number of requests in NCQ, is negatively affected by fairness mechanism of I/O schedulers (i.e., anticipation [18]) in the presence of workloads with deceptive idleness[2] — workloads with low request arriving speed (e.g., synchronous I/O applications). As a result, the performance

[1]**Aggressive workload:** A workload is aggressive when it occupies the majority of bandwidth in a concurrent environment.

[2]**Deceptive idleness:** A workload that dispatches the next request shortly after receiving the result of the previous one may temporarily appear to be idle to the I/O scheduler.

of SSD degrades.

In an effort to improve I/O fairness and SSDs' utilization, we propose a NCQ-aware I/O scheduling scheme, NASS. The basic idea of *NASS* is to elaborately control the request dispatch of workloads to relieve NCQ conflict and improve NCQ utilization at the same time. To do so, *NASS* builds an evaluation model to quantify important features of the workload. In particular, the model first finds aggressive workloads, which cause NCQ conflict, based on the request size and the number of requests of the workloads. Second, it evaluates merging tendency of each workload, which may affect the bandwidth and cause NCQ conflict indirectly, based on request merging history. Third, the model identifies workloads with deceptive idleness, which cause low NCQ utilization, based on historical requests in I/O scheduler. Then, based on the model, *NASS* sets the request dispatch of each workload to guarantee fairness and improve device utilization: (1) *NASS* limits aggressive workloads to relieve NCQ conflict; (2) it adjusts merging of sequential workloads to improve bandwidth of the workloads while relieving NCQ conflict; and (3) it restricts request dispatch of I/O scheduler, rather than stopping request dispatch to improve NCQ utilization.

We integrate *NASS* into four state-of-the-art I/O schedulers including CFQ, BFQ [19], FlashFQ [11], and FIOPS [14]. The experimental results show that with NASS, I/O schedulers can achieve 11-23% better fairness and at the same time improve device utilization by 9-29%. We also show that *NASS* introduces a small CPU overhead of 2%.

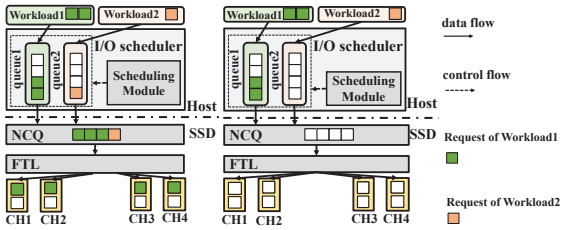
In summary, this paper makes the following contributions:

1. We comprehensively analyze how and why NCQ impacts I/O fairness and device utilization on conventional SSD (Section II).
2. We propose a NCQ-aware I/O scheduling scheme named NASS, which elaborately controls the request dispatch of I/O scheduler according to NCQ status and workload features (Section III).
3. We integrate NASS into four state-of-the-art I/O schedulers and evaluate its effectiveness in offering better fairness and higher utilization of SSD using widely used benchmarks and real world applications (Section IV).

II. Background and Motivation

A. Native Command Queueing

Native Command Queueing (NCQ) [20] is a technology introduced to SATA II and originally designed to reduce seek time of HDDs by reordering the received commands. Generally, a device with NCQ can store 32 requests by default. NCQ helps SSD to distribute requests across channels in batches and process the requests simultaneously. Therefore, the parallelism of SSD can be fully used [13]. As shown in Figure 2(a), at first, the scheduling module chooses requests to dispatch. Then, the I/O scheduler dispatches the requests to NCQ. Finally, selected requests are dispatched to FTL concurrently. The maximum number of requests that an I/O scheduler can dispatch to NCQ depends on the NCQ length which is the only configurable NCQ parameter.



(a) NCQ conflict (b) Low NCQ utilization

Fig. 2: Effect of NCQ

B. NCQ Conflict: Occurrence and Impact

NCQ conflict corresponds to the situation when requests of different workloads cannot have an equal chance to enter NCQ. For example, assume that we have a SSD with 4 channels and that NCQ length is set to 4, as shown in Figure 2(a). Two concurrent workloads are running: *Workload1* dispatches requests at high speed and *workload2* dispatches requests at low speed. I/O fairness is provided by assigning equal scheduling time (time slice [18], [11]) to each workload. Here, NCQ will be filled with requests from *Workload1*, due to its high-speed arrival and consequently requests from *Workload2* will wait and cannot be handled by the SSD on time. As a result, NCQ conflict occurs and obviously, fairness is violated.

As a mean to handle NCQ conflict, I/O schedulers are equipped with anticipation technique. As Figure 2(b) shows, when requests of *workload2* are anticipated, the scheduling module stops scheduling requests for a short time to wait for requests of *workload2*. This prevents *workload1* from occupying the whole NCQ and allows requests of *workload2* to be dispatched and handled on their arrival. But, as requests of *workload1* cannot be dispatched even though NCQ is free, the performance of SSD decreases. In fact, restricting the request dispatch of I/O scheduler, rather than stop scheduling, allows the SSD to keep serving requests of *workload1* as well as reserve resource for *workload2* (as discussed in Section II-C).

To conclude, there is no coordination between I/O schedulers which target fairness and request dispatch in NCQ and thus NCQ conflict may happen. Even worse, optimizations at the I/O scheduler level to handle NCQ conflict may lower the utilization of NCQ (requests cannot enter NCQ even when it is free). This may impact the fairness and utilization of the SSDs negatively. Hereafter, we provide an in-depth analysis of fairness and resource utilization in SSDs using four state-of-the-art fairness driven I/O schedulers and when varying NCQ length.

It is important to mention that NCQ conflict is specific to SSD and does not really occur in HDDs. NCQ is used in HDDs to reduce seek time by reordering the requests to improve the bandwidth of random workloads, and to improve the bandwidth of sequential workloads by enhancing merging [17]. However, given that requests from one workload cannot occupy the whole NCQ in HDDs, HDDs show no NCQ conflict: HDD handles requests in low speed and therefore I/O scheduler can only issue a few requests to NCQ in one scheduling share.

TABLE I: State-of-the-art fairness driven I/O schedulers

I/O Scheduler	Description	Anti	Ideal Fairness	SF
CFQ [18]	CFQ is default I/O scheduler of CentOS. It achieves fairness by assigning equal time slice to each workload to dispatch requests.	optional	Each workload occupies equal device time.	low
BFQ [19]	BFQ has been merged for Linux 4.12. It is based on CFQ, but it provides fairness by assigning equal sector quota to each workload.	optional	Each workload occupies equal bandwidth.	low
FlashFQ [11]	FlashFQ is based on SFQ(D) [21]. It dispatches requests according to the arrival time and processing time of requests from different workloads.	yes	Each workload occupies equal device time.	high
FIOPS [14]	FIOPS is a new Linux I/O scheduler for SSD. It is similar to CFQ, while the request dispatch decision on FIOPS is based on IOPS instead of time. FIOPS only issues one request to SSD in a scheduling period.	no	Each workload occupies equal IOPS.	high
CFQ and BFQ without anticipation are called CFQ-NA and BFQ-NA, respectively. **Anti** is anticipation, **SF** is scheduling frequency.				

C. Empirical Evaluation of the Impact of NCQ

To better understand how NCQ affects I/O fairness and utilization of SSDs, we carry out extensive experiments when varying NCQ length which impacts the number of requests that a SSD can handle at a time and affects the total number of dispatched requests of I/O scheduler. We use four state-of-the-art I/O schedulers including CFQ, BFQ, FlashFQ, and FIOPS. They all target fairness based on different principles. For example, while fairness in CFQ is achieved by assigning equal time slice to each workload, BFQ targets fairness by assigning equal sector quota to each workload. More details about the studied schedulers are presented in Table I.

1) Experimental Setup: The physical node has four quad-core 2.4GHz Xeon processors. CentOS 6.5 and Linux 3.10.107 kernel are used. I/O workloads with different I/O patterns are generated by FIO [22]. We set the *Iodepth* of FIO (i.e., the number of requests which can be issued to the lower-layer of the system at a time) to 32 and 2 for intensive and non-intensive workloads, respectively. Direct I/O is used to avoid the effect of memory caching. We use three kinds of SSDs from different vendors: (1) An Intel 530 SSD; (2) A Huawei OceanStor SSD; and (3) A Toshiba A100 SSD.

Fairness Metric: We use Jain's fairness measure to quantify the fairness [23]. It is a widely used metric to quantitatively measure fairness in shared computer systems [24–26]:

$$FV = \frac{(\sum_{i=1}^{m} P_i)^2}{m \times \sum_{i=1}^{m} P_i^2} \quad (1)$$

where P_i is the resources that *workload i* gets from the SSD per second. The fairness value (FV) ranges between 0 (no fairness) and 1 (perfect fairness). Note that *P* corresponds to different measures according to the evaluated I/O schedulers. In particular, BFQ aims to achieve fair bandwidth sharing, accordingly *P* is the obtained bandwidth of a workload. FIOPS aims to achieve fair IOPS sharing, so *P* is the achieved IOPS of a workload.

CFQ and FlashFQ aim to achieve fair time slice sharing. So *P* is the *device time occupation per second* (*TOPS*) of a workload. However, *TOPS* is hard to measure directly. We use achieved bandwidth to calculate P of *workload i*:

$$P_i = BDR_i = \frac{B_i}{B_{ialone}} \quad (2)$$

where *bandwidth degradation ratio* of *workload i* (BDR_i) is the ratio of B_i to B_{ialone}. B_{ialone} is the bandwidth when *workload i* is running alone and B_i is the bandwidth when *workload i* is running in a concurrent environment. Note that when *workload i* is running alone, it occupies the whole device time. Thus, the achieved bandwidth is B_{ialone} when $TOPS_i$ is equal to 1. When *workload i* is running in a concurrent environment, it achieves a bandwidth B_i proportional to its $TOPS_i$. Accordingly, $\frac{B_i}{B_{ialone}} = \frac{TOPS_i}{1}$ and $TOPS_i$ is equal to BDR_i. This explains why we use *BDR* to represent *P* for CFQ and FlashFQ.

TABLE II: Workload configurations. We have eight groups of tests. In these tests, different workloads are concurrent.

	workload1(W1)	workload2 (W2)
TEST1	4K sequential read	4K random read
TEST2	4K sequential write	4K random write
TEST3	4K sequential read	4K sequential write
TEST4	4K random read	4K random write
TEST5	4K random read	128K random read
TEST6	4K random write	128K random write
TEST7	4K non-intensive read	4K intensive read
TEST8	4K non-intensive write	4K intensive write

Device Utilization Metric: *Device utilization* (*DU*) measured in the system layer essentially indicates the percentage of time that the device is busy serving at least one request [27]. This is true for HDD, because HDD only handles one request at a time. For SSD, which can handle multiple requests at a time, the measurement is not accurate. Therefore, similar to Shen [11], we define *DU* of SSD as the sum of all workloads' device time occupations t_i. Suppose there are *n* workloads, *DU* should be:

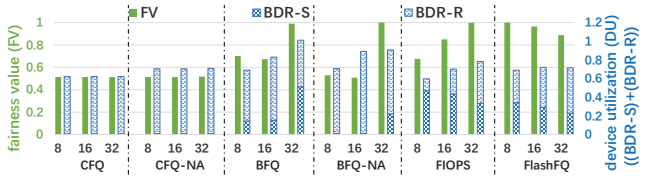
$$DU = \sum_{i=1}^{n} t_i = \sum_{i=1}^{n} \frac{B_i}{B_{ialone}} = \sum_{i=1}^{n} BDR_i \quad (3)$$

DU is a higher-is-better metric. It may be lager than 1, if parallelism of SSD is fully exploited.

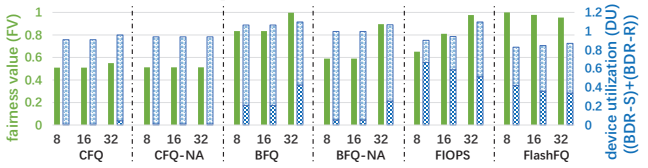
Workloads: The specific parameters of each workload are listed in Table II. Two concurrent workloads are running in each test. They have different sequentiality (TEST1 and TEST2), read/write features (TEST3 and TEST4), request sizes (TEST5 and TEST6), and intensities which stands for arriving speed of requests (TEST7 and TEST8). We mainly use workloads with small size requests (4K) for comparison, because the variations of I/O patterns have more obvious impact on small size requests compared to big ones.

2) Experimental Results: NCQ length limits the maximum number of requests in NCQ, thus controlling the request dispatch of I/O scheduler. So NCQ length is set to 8, 16, and 32 respectively to observe how NCQ affects fairness and device utilization under concurrent environments. Notably, the experimental results of three types of SSDs show similar trends, so we give our analyses based on Intel SSD. Results on Toshiba and Huawei are available here [28].

Workloads with Different Sequentialities: Under default NCQ length (32), the bandwidth of the SSD is occupied by random workload in CFQ and CFQ-NA as shown in Figure 3, because of the merging operation in I/O scheduler



(a) Results when sequential read and random read are concurrent



(b) Results when sequential write and random write are concurrent

Fig. 3: Fairness value and device utilization when workloads with different sequentiality are concurrent. NCQ lengths are 8, 16, and 32. The left y-axis stands for fairness value. The right y-axis stands for device utilization. Device utilization is the sum of BDR of workloads. BDR-S stands for BDR of the workload with sequential requests. BDR-R stands for BDR of the workload with random requests.

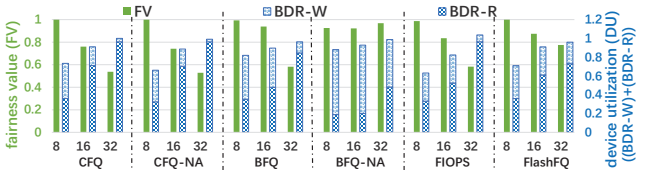
[17]. Requests with good sequentiality are merged, and the merging reduces the number of requests of sequential workload. As a result, I/O scheduler can only dispatch requests of random workload when sequential workload runs out of requests and waits for requests from upper-layer system. This makes random requests occupy most of the NCQ, and requests of sequential workload cannot be handled during its next scheduling share. Consequently, NCQ conflict occurs, and the bandwidth of sequential workload degrades. Sequential workload gets high bandwidth under FIOPS and FlashFQ. High scheduling switch frequency of FlashFQ and FIOPS makes requests of sequential workload get more chances to be scheduled. This reduces request merging of the workload. Consequently, NCQ conflict is alleviated.

As NCQ length increases, for BFQ and BFQ-NA, fairness is improved. The reason lies in that with the increase of NCQ length, more requests in I/O scheduler can be dispatched and the impact of request merging of sequential workload is weakened. Thus NCQ conflict is alleviated. For FIOPS, as NCQ length increases, the bandwidth of sequential workload degrades, because decreased merging reduces request size [17]. As a result, fairness is improved.

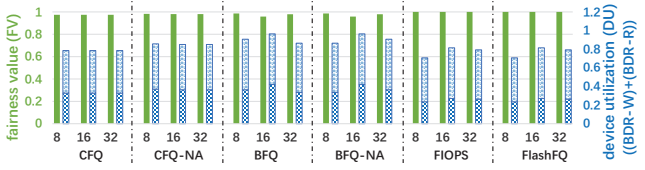
Observation 1: Request merging aggravates NCQ conflict, because it reduces the number of requests of sequential workload. This makes requests of random workload occupy most of NCQ when I/O schedulers have low scheduling switch frequency. As a result, unfairness happens.

Workloads with Different Read/write Features: Figure 4 shows the TESTs when workloads are with different read/write features. For TEST3 in Figure 4(a), I/O schedulers favor read workload under default NCQ length. This is due to the asymmetric of read/write [29], which means as process time of read is shorter than that of write, read requests can be handled at high speed. This leads to more read requests entering NCQ, and lowers the dispatch of write requests. Moreover, merging in I/O scheduler reduces the number of write requests. Consequently, read requests occupy most of the NCQ and the bandwidth of write workload degrades.

As NCQ length decreases, the total number of dispatched requests of read workload goes down. This makes read



(a) Results when sequential read and write are concurrent



(b) Results when random read and write are concurrent

Fig. 4: Fairness value and device utilization when workloads with different read/write features are concurrent. BDR-W stands for BDR of the workload with write requests. BDR-R stands for BDR of the workload with read requests.

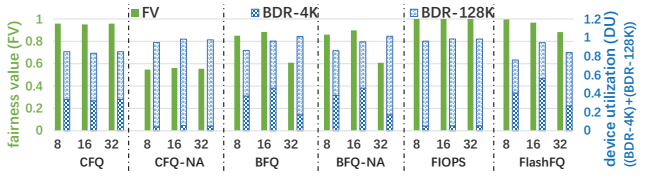
requests merge into big size requests like write. As the process time of big size read and write become close [29], NCQ conflict caused by read/write asymmetric is relieved and fairness is improved as shown in Figure 4(a).

Observation 2: Read/write asymmetric causes NCQ conflict, because it throttles down write requests, and makes them merge into big requests. As a result, the number of write requests decreases and read requests occupy most of NCQ. This causes unfairness.

Workloads with Different Request Sizes: Figure 5 shows the TESTs when workloads are with different request sizes. Under default NCQ length, BFQ and BFQ-NA fail to achieve high fairness because of NCQ conflict caused by request size difference. The principle of equal bandwidth share makes BFQ and BFQ-NA dispatch more 4K requests. When there is no small size request to dispatch, BFQ and BFQ-NA will dispatch big size requests. In SSD, a big size request is translated into several sub-requests that occupy several channels. As a result, 4K requests suffer from NCQ conflict.

Decreasing NCQ length reduces the number of dispatched requests of I/O scheduler. The dispatch of 128K requests is limited, and NCQ conflict is relieved. Therefore, BFQ and FlashFQ get better fairness when NCQ length decreases.

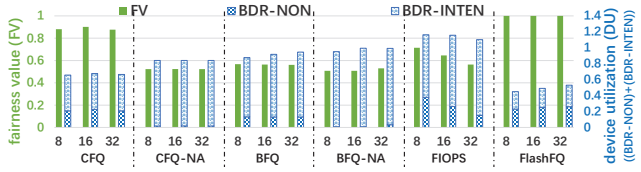
Observation 3: Workloads with big request sizes cause NCQ conflict, because a big size request occupies more SSD channels. When there are big size requests in NCQ beyond SSD's capacity, small size requests of other workloads



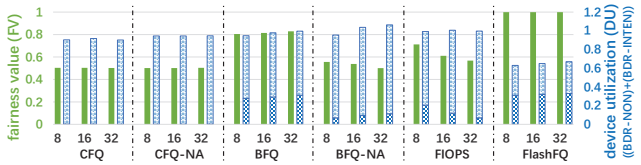
(a) Results when 4K random read and 128K random read are concurrent

(b) Results when 4K random write and 128K random write are concurrent

Fig. 5: Fairness value and device utilization when workloads with different request sizes are concurrent. BDR-4 stands for BDR of the workload with 4K requests. BDR-128 stands for BDR of the workload with 128K requests.



(a) Results when non-intensive read and intensive read are concurrent



(b) Results when non-intensive write and intensive write are concurrent

Fig. 6: Fairness value and device utilization when workloads with different intensities are concurrent. BDR-NON stands for BDR of the non-intensive workload. BDR-INTEN stands for BDR of the intensive workload.

cannot be handled. This causes unfairness.

Workloads with Different Intensities: Figure 6 shows workloads with different I/O intensities. Under default NCQ length, BFQ-NA, CFQ-NA, and FIOPS fail to achieve fairness. In these schedulers, non-intensive workload cannot dispatch requests to I/O scheduler in time. NCQ conflict happens, because requests from intensive workload occupy the whole NCQ, and requests of non-intensive workload cannot be issued to SSD when they reach NCQ. CFQ and FlashFQ have relatively better fairness at the cost of bandwidth loss because of the waiting for caused by anticipation. Notably, write tests in CFQ show low fairness, because CFQ does not provide anticipation for write requests.

As NCQ length decreases, the number of dispatched requests of intensive workload goes down, and requests from non-intensive workload can be dispatched on their arrival due to high scheduling switch frequency of FIOPS. As a result, FIOPS shows improved fairness.

Observation 4: Intensive workload causes NCQ conflict, because intensive workload sends excessive requests to NCQ beyond SSD's capacity, and requests of non-intensive workload cannot be handled in time. This causes unfairness.

Observation 5: Results of different NCQ lengths show that limiting the request dispatch of aggressive workloads (random workload in TEST1 and 2, read workload in TEST3, intensive workload in TEST7 and 8) can relieve NCQ conflict. However, when NCQ length is too short, device utilization degrades because of low NCQ utilization.

Observation 6: By comparing CFQ and CFQ-NA in all tests, we find that anticipation lowers NCQ utilization. This is because anticipation forbids request dispatch of other workloads when waiting for requests of a workload with deceptive workload. Consequently, device utilization reduces.

III. NCQ-aware I/O Scheduling Scheme

Based on the observations and in an effort to improve I/O fairness and improve SSDs' utilization, we propose a NCQ-aware I/O scheduling scheme, *NASS*. Hereafter, we will first summarize the design principles of *NASS* and then we focus on the design details of *NASS*.

A. Design Principles of NASS

NASS is designed with the following goals in mind:

- **Relieve NCQ conflict:** This is critical to improve I/O fairness, especially when workloads with different I/O patterns are running concurrently. NCQ conflict occurs when a SSD is shared with aggressive workloads (i.e, workloads with relatively high request arriving speed or big requests). Consequently, requests of a non-aggressive workload cannot enter SSD and fairness is violated (**Observation 3** and **4**). *NASS* aims to reduce the impact of NCQ conflict by limiting request dispatch of aggressive workloads (**Observation 5**).
- **Control request merging:** Request merging can improve bandwidth when a sequential workload is running alone. However, merging may aggravate NCQ conflict and cause unfairness indirectly, because unlimited merging reduces the number of requests and turns the workload into a non-aggressive workload. As a result, unfairness happens (**Observation 1** and **2**). *NASS*, therefore, adjusts request merging to keep improving the SSD utilization without causing NCQ conflict.
- **Handle deceptive idleness:** Current I/O schedulers employ anticipation to handle workloads with deceptive idleness. Anticipation reserves resource for the workload with deceptive idleness which reduces NCQ utilization and device utilization (**Observation 6**). Instead of using anticipation, *NASS* reserves resources by restricting the request dispatch of the I/O scheduler to improve NCQ utilization.
- **I/O scheduler independent:** *NASS* can be built on the top of any I/O scheduler, simply by adding a dispatch control module.

To achieve these goals, first, *NASS* builds an evaluation model to quantify workloads. The evaluation model can find aggressive workloads which cause NCQ conflict, perceive sequential workloads which affect NCQ conflict indirectly and identify workloads with deceptive idleness which lowers NCQ utilization. Then, based on the model, *NASS* limits request dispatch of aggressive workloads to relieve NCQ conflict, adjusts merging of sequential workloads to improve the bandwidth of the workloads without causing NCQ conflict and restricts request dispatch of I/O scheduler to improve NCQ utilization. By integrating *NASS* into an existing I/O scheduler, NCQ conflict and NCQ low utilization can be addressed effectively which in turn results in improving both I/O fairness and utilization of SSDs.

B. Workload Evaluation Model

In order to find workloads that cause NCQ conflict or low NCQ utilization, *workload evaluation model* (WEM) needs to evaluate a workload in three aspects: (1) whether the workload is aggressive, (2) whether the workload causes merging, and (3) whether the workload has deceptive idleness. Notalby, Table III lists variables used in Section III.

1) Aggressiveness Evaluation: For a workload, W_i, the aggressiveness of W_i, $Aggre(W_i)$, represents how many channels it requires to handle all dispatched requests of W_i at

TABLE III: Variables of the workload evaluation model

Var.	Description	Var.	Description	Var.	Description
W_i	a workload	$Size(R_i^j)$	the size of R_i^j	$W(n)$	workload number in I/O scheduler
α	a constant	$S(W_i, T)$	the sequentiality of W_i	$Aggre(ave)$	average aggressiveness of all workloads
R_i^j	a request of W_i	$Aggre(W_i)$	the aggressiveness of W_i	$Cn(R_i^j)$	the number of channels a request needs
μ	a decay factor	NCQ_{def}	default NCQ length (32)	$W(R_i^j, T)$	weight of R_i^j in calculating sequentiality
T	the current time	$W(ssd)$	workload number in SSD	m	the number of requests from W_i until time T_l
T_r	R_i^j accesses in T_r	$Size(sub)$	the sub-request size of SSD	$MDN(W_i)$	the maximum number of dispatched requests of W_i
θ	a factor can be -1 or 1	T_l	the arrival time of the last request	$WC(h)$	workload collection of α historical requests
η	channel number of SSD	$Haggre(W_i)$	the history aggressiveness of W_i	$MDN(total)$	the maximum number of dispatched requests of scheduler

a time (i.e., requests which are dispatched by the I/O scheduler but are still not completed by the SSD). For example, if sixteen 4K requests of W_i are dispatched, 16 channels are required to handle these requests. So $Aggre(W_i)$ is 16. We consider two factors in calculating $Aggre(W_i)$:

- **Request Size:** A request of big size requires more channels, because it is divided into several sub-requests. Each sub-request will be then sent to a distinct channel.
- **Number of dispatched requests:** Ideally, requests are sent to different channels. So the more requests there are, the more channels they require.

Note that read/write feature and merging are not considered, because they cause NCQ conflict indirectly by changing the size and number of requests.

First, we calculate $Cn(R_i^j)$ which stands for the number of channels required by a request, R_i^j, of W_i. When request size of R_i^j, $Size(R_i^j)$, is bigger than the size of sub-request (the maximum size of a (sub)request which one channel can serve), $Size(sub)$, R_i^j is divided into several sub-requests with size of $Size(sub)$, and $Cn(R_i^j)$ is equal to the number of sub-requests. When $Size(R_i^j)$ is not bigger than $Size(sub)$, it is unnecessary to divide R_i^j and $Cn(R_i^j)$ is equal to one. So the number of channels a request needs, $Cn(R_i^j)$, can be calculated as:

$$Cn(R_i^j) = \begin{cases} 1 & Size(R_i^j) \leq Size(sub) \\ \lfloor \frac{Size(R_i^j)}{Size(sub)} \rfloor + 1 & Size(R_i^j) > Size(sub) \end{cases} \quad (4)$$

Second, we calculate $Aggre(W_i)$ which is the number of channels required by all dispatched requests of W_i. $Aggre(W_i)$ can be written as:

$$Aggre(W_i) = \sum_{R_i^j \in R(W_i)} Cn(R_i^j) \quad (5)$$

where $R(W_i)$ is all the dispatched requests of W_i. The bigger $Aggre(W_i)$ is, the more aggressive W_i is. W_i causes NCQ conflict if $Aggre(W_i)$ is bigger than average aggressiveness of all workloads, $Aggre(ave)$.

2) Sequentiality Perception: Sequentiality, which indicates the merging tendency of a workload, is the sum of contributions of all historical requests of the workload.

We introduce a weight to the request when arriving at the I/O scheduler, $W(R_i^j, T)$, to calculate its contribution to sequentiality. Factor θ is used to show whether the contribution of R_i^j is positive or negative. When R_i^j can merge with a request in I/O scheduler, θ is set to 1, because R_i^j has a positive contribution to the sequentiality of W_i, $S(W_i, T)$. When R_i^j cannot merge, θ is set to -1, because R_i^j has a negative contribution to $S(W_i, T)$. In addition, the contributions of historical requests decline with time, as the I/O pattern of a workload changes with time. A decay factor, μ, is used in the weight to quantize the relationship between time and the contribution of the request. Weight of R_i^j, $W(R_i^j, T)$, is:

$$W(R_i^j, T) = \theta \mu^{-(T - T_r)} \quad (6)$$

where $\mu > 1$, T_r is the arrival time of R_i^j at the I/O scheduler, and T is the current time. According to Equation (6), a newly arrived request has more impact on the sequentiality of W_i compared to a past request.

Accordingly, when a new request R_i^m of a workload W_i arrives at the I/O scheduler at T_n, sequentiality of W_i at T_n, $S(W_i, T_n)$, is given as:

$$\begin{aligned} S(W_i, T_n) &= \sum_{k=1}^{m} W(R_i^k, T_n) \\ &= \sum_{k=1}^{m-1} W(R_i^k, T_n) + \sum_{k=m}^{m} W(R_i^k, T_n) \\ &= \mu^{-(T_n - T_l)} S(W_i, T_l) + \theta \mu^{(T_n - T_n)} \\ &= \mu^{-(T_n - T_l)} S(W_i, T_l) + \theta \end{aligned} \quad (7)$$

*w*here R_i^k represents the k^{th} request from W_i, and T_l is the arrival time of the last request. $S(W_i, T)$ consists of the weight of the current request and the decay of sequentiality of the previous requests. When $S(W_i, T)$ is bigger than 0, we consider that merging occurs frequently in W_i. When W_i merges frequently, it may affect NCQ status.

3) Deceptive Idleness Identification: Accurate deceptive idleness identification is hard, because I/O scheduler has no information of the upper-level system. Furthermore, as request dispatch restriction of I/O scheduler shows a slight resource waste when there is a wrong identification, we use a simple method to identify deceptive idleness.

WEM triggers *deceptive idleness identification* for W_i when there is no request of W_i in I/O scheduler. *WEM* considers W_i as workload with deceptive idleness if there are requests of W_i in past α historical requests, because there may be requests of W_i coming to I/O scheduler soon. While if there is no request of W_i in past α historical requests, *WEM* considers W_i is terminated, because W_i has no request arrived for a long time.

C. Dispatch Control Algorithm

Based on the results of *WEM*, *dispatch control algorithm* (*DCA*) controls request dispatch. Request dispatch control is realized by setting the number of dispatched requests. *DCA* controls request dispatch in three steps. First, based on the sequentiality of a workload, request dispatch of sequential workload is adjusted to enhance merging and improve the bandwidth of the workload. If the adjustment causes NCQ conflict because of too much merging, it is corrected in the third step. Second, when there is a workload with deceptive idleness, the request dispatch of I/O scheduler is restricted

Algorithm 1 Dispatch Control Algorithm

/*DAC is triggered when a request arrives at I/O scheduler, enters SSD or is completed.*/

Input: $Aggre(ave)$; $WC(h)$; $W(ssd)$; $Haggre(W_i)$; R_i^j; $S(W_i,T)$; $Cn(R_i^j)$; $W(n)$; $Aggre(W_i)$; η; NCQ_{def}

Output: $MDN(total)$; $MDN(W_i)$

1: **if** R_i^j *arrives at I/O scheduler* **then**
2: *Calculate* $S(W_i,T)$
3: **if** W_i *is a new workload* **then**
4: $Aggre(W_i) = 0$
5: MDN *of workloads are set to* $NCQ_{def}/W(n)$
6: **end if**
7: **if** $S(W_i,T) > 0$ **then**
8: **if** $Haggre(W_i) \leq Aggre(W_i)$ **then**
9: *Decrease* $MDN(Wi)$
10: **else**
11: *Increase* $MDN(Wi)$
12: **end if**
13: **end if**
14: **end if**
15: **if** R_i^j *enters SSD* **or** R_i^j *is completed* **then**
16: **if** R_i^j *enters SSD* **and** W_i *has no request in I/O scheduler* **then**
17: **if** $W_i \in WC(h)$ **then**
18: *Restrict* $MDN(total)$
19: **else**
20: *Set* $MDN(total)$ *to* NCQ_{def}
21: **end if**
22: **end if**
23: $Haggre(W_i) = Aggre(W_i)$
24: *Update* $Aggre(W_i)$
25: **if** $Aggre(total) > \eta$ **and** $W(ssd) > 1$ **then**
26: **if** $Aggre(W_i) > Aggre(ave)$ **then**
27: *Set* $MDN(Wi)$ *to* $MDN(Wi)/2$
28: **end if**
29: **end if**
30: **end if**

to reserve NCQ for the workload. Third, according to the number of dispatched requests of I/O scheduler in step 2, request dispatch of aggressive workloads are limited to relieve NCQ conflict. As Algorithm 1 shows:

Lines 1-14 calculate sequentiality and carry out *request dispatch adjustment based on sequentiality*. When W_i sends a request, R_i^j, to I/O scheduler, $S(W_i,T)$ is calculated based on merging history in the I/O scheduler. If W_i is a new workload, $Aggre(W_i)$ is set to 0 and the maximum number of dispatched requests of each workload is set to the quotient of default NCQ length and number of workloads, $NCQ_{def}/W(n)$. When W_i has good sequentiality ($S(W_i,T) > 0$), *DCA* judges whether merging is necessary based on the change of aggressiveness. *DCA* compares $Haggre(W_i)$, which is the $Aggre(W_i)$ before update (i.e., the arrival of a new request), with $Aggre(W_i)$ when a new request of W_i arrives at the I/O scheduler. If $Aggre(W_i)$ is greater than $Haggre(W_i)$ merging can improve bandwidth by increasing request sizes. So *DCA* gradually decreases the maximum number of dispatched requests of W_i, $MDN(W_i)$, to enhance merging. If $Aggre(W_i)$ is smaller than $Haggre(W_i)$ – some of the earlier dispatched requests are completed by the SSD – merging will not lead to an improvement in the bandwidth as SSD will not be able to handle more requests or there are not enough requests issued by the workload. Moreover, unfairness occurs because of the decrease in aggressiveness. So *DCA* gradually increases $MDN(W_i)$ to reduce merging.

Lines 15-22 show *request dispatch restriction of I/O scheduler based on deceptive idleness*. When there is no

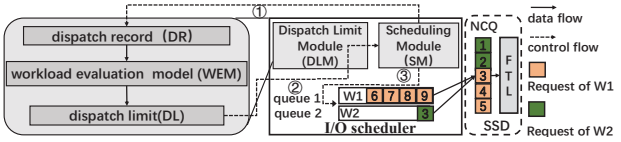


Fig. 7: Overview of NCQ-aware I/O scheduler

request of W_i in the I/O scheduler, *DAC* judges whether W_i is a workload with deceptive idleness. If there are requests of W_i in historical requests, $W_i \in WC(h)$, W_i is a workload with deceptive idleness. $MDN(total)$ is restricted to β to reserve resources for W_i.

Lines 23-30 show *request dispatch limit based on aggressiveness*. When a request of W_i enters SSD or is completed, $Aggre(W_i)$ is calculated based on requests in SSD. Dispatch limit is triggered when requests in SSD are from different workloads, and more than η (total channel number of SSD) channels are required to handle all these requests at a time. If aggressiveness of W_i is larger than $Aggre(ave)$, W_i causes NCQ conflict. We set $MDN(W_i)$ to $MDN(Wi)/2$ to prevent excessive dispatch. If W_i is still aggressive, $MDN(W_i)$ is further limited when a new request arrives.

D. Implementation of NASS

Figure 7 shows the overview of *NASS*. *NASS* consists of three parts: 1. *Queues* for holding requests from different workloads. 2. *Scheduling module* (*SM*) for choosing a queue to dispatch requests. 3. *Dispatch control module* (*DCM*) for controlling the request dispatch of queues. *NASS* works as follows: ① *SM* selects a queue to dispatch a request and informs *DCM*. *DCM* makes a judgment based on records in *dispatch record*. ② If the number of dispatched requests of the queue exceeds the limit, *DCM* informs the *SM* to reselect a queue to dispatch a request. ③ Or *SM* dispatches requests of the chosen queue to NCQ. *DCM* is the key difference between *NASS* and traditional scheduling scheme. *Dispatch control algorithm* (*DCA*) is realized in this module.

Parameter Settings: The *workload evaluation model (WEM)* includes quite a few parameters. Channel number of Intel SSD, η, is 10. Sub-request size, $Size(sub)$, of the SSD equals to the page size (8KB) of SSD [30]. The historical number of requests, α, is equal to the default I/O scheduler queue length (128). Decay factor, μ, decides contributions of historical requests to merging tendency. Like Pregather [31, 32], we set it to 2 by default. Total number of dispatched requests of I/O scheduler, β, determines the parallelism of SSD. Figure 8 shows the device utilization when an intensive workload is running alone. On the one hand, β cannot be too small, because small β lowers the device utilization. On the other hand, β cannot be too big, because big β lowers fairness. In this paper, β is set to 8, where the workload shows obvious performance degradation for the first time. This makes sure that requests of a workload with deceptive

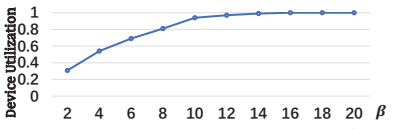


Fig. 8: Device utilization under different βs

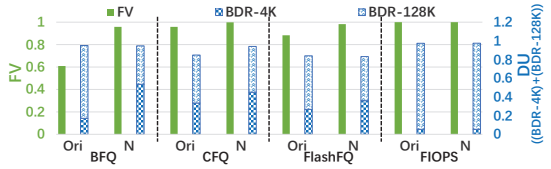


Fig. 9: Tests when workloads are with different request sizes. Ori is original I/O scheduler under default NCQ length, N is NQC-aware I/O schedulers. The left y-axis stands for fairness value (FV). The right y-axis stands for device utilization (DU). DU is the sum of BDR of workloads. BDR-4K is BDR of workload with 4K requests. BDR-128K is BDR of workload with 128K requests.

idleness can be issued on their arrival without preventing other workloads from issuing requests.

IV. Experimental Evaluation

We integrate *NASS* into four state-of-the-art I/O schedulers and realize BFQ(N), CFQ(N), FlashFQ(N), and FIOPS(N). We run a suite of experiments to compare our NCQ-aware I/O schedulers with the original I/O schedulers. As shown in the empirical analysis (II-C2), short NCQ length shows a slight improvement in fairness in most cases and causes degradation in device utilization, so we set NCQ length to 32 in the experiments. First, we evaluate *NASS* internals: the *workload evaluation model* (*WEM*) and *dispatch control algorithm* (*DCA*). Second, we evaluate the effectiveness of *NASS* using real world applications. Third, we evaluate the overhead caused by *NASS*. Experimental settings are the same as the ones described in Section II-C1. We discuss the results on Intel SSD only, however the results on Toshiba and Huawei SSDs are available here [28].

A. Micro-benchmark Results: NASS Internals

1) Dispatch Limit Based on Aggressiveness: Figure 9 shows the results when 4K random read (non-aggressive workload) and 128K random read (aggressive workload) are running concurrently (TEST5). Both FIOPS and FIOPS(N) show high fairness and high device utilization. This means that dispatch limit does not have a side effect on FIOPS. Compared with the original I/O schedulers, FVs of CFQ(N), FlashFQ(N), and BFQ(N) show 4%, 11%, and 57% improvement, respectively. This is because WEM can detect NCQ conflict caused by workloads with big request sizes, and request dispatch limit based on aggressiveness can limit request dispatch of those workloads.

As for DU, both original and modified I/O schedulers show high performance, because anticipation has little influences in this scenario.

2) Dispatch Restriction of I/O Scheduler Based on Deceptive Idleness: Figure 10 shows the results when 4K non-intensive and 4K intensive read are running concurrently (TEST7). CFQ and FlashFQ show low DUs, because of long idling time caused by anticipation. By restricting request dispatch of I/O scheduler instead of anticipation, DUs of CFQ(N) and FlashFQ(N) show 59% and 70% improvement,

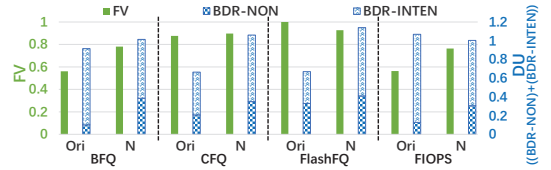


Fig. 10: Tests when workloads are with different intensities. BDR-NON is BDR of non-intensive workload. BDR-INTEN is BDR of intensive workload.

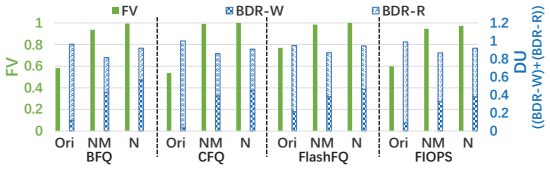


Fig. 11: Tests when sequential workloads are with different read/write features. Ori is the original I/O scheduler under default NCQ length, NM is *NASS* that without *SBDA* and N is full function *NASS*. BDR-W is BDR of workload with write requests. BDR-R is BDR of workload with read requests.

respectively. The reason is that request dispatch restriction allows more requests to enter NCQ compared to anticipation. DU of BFQ(N) shows 11% improvement, because idling time caused by anticipation of BFQ is short and only has little influence on DU. Notably, DU of FIOPS(N) shows a slight degradation, because FIOPS has no anticipation.

CFQ has high fairness, so FV of CFQ(N) shows only 3% improvement. On the contrary, FVs of BFQ(N) and FIOPS(N) are improved by 40% and 35%, respectively. The reason is that *WEM* can identify non-intensive workload which causes low NCQ utilization because of deceptive idleness, and request dispatch restriction of I/O scheduler based on deceptive idleness can reserve resources for non-intensive workload and thus requests of non-intensive workload can be dispatched on their arrival. As a result, fairness is improved. For FlashFQ(N), FV is decreased compared to FlashFQ. But FlashFQ(N) results in higher device utilization (the *BDR*s of two workloads under FlashFQ(N) are improved).

3) Dispatch Adjustment Based on Sequentiality: Figure 11 shows the results when 4K sequential read and write are running concurrently (TEST3) under the original I/O schedulers, *NASS* without *sequentiality-based dispatch adjustment* (*SBDA*), and *NASS* with *SBDA*, separately. All original I/O schedulers show poor fairness because of the NCQ conflict caused by read/write asymmetric. For *NASS* without *SBDA*, FV is improved by 30% (FlashFQ) at least, because of dispatch limit based on aggressiveness. Original I/O schedulers show higher DU compared to *NASS* without *SBDA*, because I/O resource reserved by read workload cannot be fully used by write workload.

Compared with *NASS* without *SBDA*, *NASS* with *SBDA* shows improvement in FV and DU. BFQ(N), CFQ(N), FlashFQ(N), and FIOPS(N) show 6%, 2%, 2%, and 3% improvement in FV, and 13%, 6%, 9%, and 12% improvement in DU, respectively. The reason is that *WEM* can detect sequential workload, and request dispatch adjustment based on sequentiality can enhance merging of write requests. As a result, device resource reserved by read workload can be used more effectively.

B. Effectiveness of NASS Using Real World Applications

1) Applications with Complex I/O Pattens: We concurrently run three widely deployed real world workloads generated by Filebench: File server, Mail server, and Database

TABLE IV: Parameters for Filebench workloads

Server	Read/write ratio	Sequentiality	Threads	Request size
File	1:2	sequential	32	16-512KB
Mail	1:1	sequential	8	16KB
DB	2:1	random	64	4KB
High thread number means high intensity.				

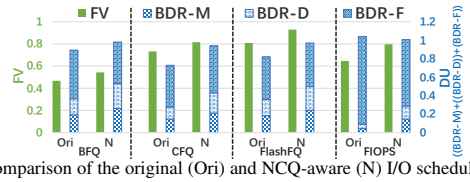


Fig. 12: Comparison of the original (Ori) and NCQ-aware (N) I/O schedulers. *BDR-M*, *BDR-D*, and *BDR-F* are BDRs of Mail server, DB server, and File server, respectively.

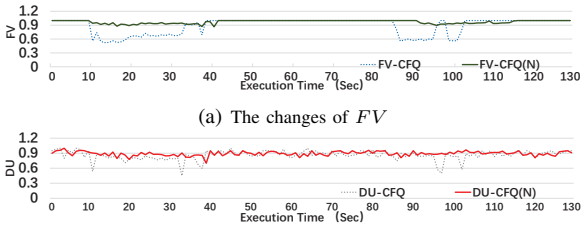
(DB) server. These workloads are different in read/write features, sequentialities, intensities, and request sizes as shown in Table IV.

Figure 12 shows the results of the original and NCQ-aware I/O schedulers when three workloads are running concurrently. File server shows high aggressiveness, which leads to unfairness in the original I/O schedulers, because File server is intensive and has big request sizes. BFQ(N), CFQ(N), FlashFQ(N), and FIOPS(N) achieve 16%, 11%, 15%, and 23% better fairness, respectively. This is because they can limit the aggressiveness of File sever, and requests from Mail server and DB server can be dispatched on time. Notably, even if BFQ(N) still results low fairness, *BDR*s of Mail server and DB server under BFQ(N) show 35% and 54% improvement, respectively.

As for DU, BFQ(N), FlashFQ(N), and CFQ(N) improve the device utilization by 9%, 18%, and 29%, respectively, because *NASS* can avoid idleness caused by anticipation. DU of FIOPS(N) shows slight degradation compared to FIOPS, because FIOPS has no anticipation, and FIOPS(N) limits request dispatch to improve fairness.

2) NASS for Big Data Analytics: We also evaluate *NASS* when SSD is shared by multiple applications including data processing applications (i.e., big data analytics on top of Spark [33]). Accordingly, we run an *OLTP* instance generated by Sysbench side by side with two representative big data workloads including *Sort* and *Wordcount*. The two workloads are from HiBench [34], a big data benchmarking suite. We run three containers with Docker-18.05.0-ce [35] on the physical host described before (Section II-C1). One container runs *OLTP*: 60 million table entries are created and 8 threads run simultaneously. We deploy single-node mode Spark on the other two containers and run *Sort* and *Wordcount*, respectively. We use Spark-2.1.1+Hadoop-2.7.3 to run big data workloads. *Sort* processes input data size of 8 GB, where 8 map tasks can run simultaneously. *Wordcount* processes 10 GB of input data and 8 map tasks can run simultaneously. Under CFQ, *OLTP* continuously issues small size requests (4KB on average in the system layer) from 0s to 130s. While, *Sort* and *Wordcount* have I/O bursts during some periods. *Sort* has intensive big size (800KB on average in the system layer) reads and writes in the map phase (during 12s to 34s) and intensive writes in the reduce phase (during 87s to 103s). *Wordcount* has lighter I/O with big request sizes (500KB on average in the system layer) in the map phase (during 20s to 39s), because it extracts a small amount of data from the input data.

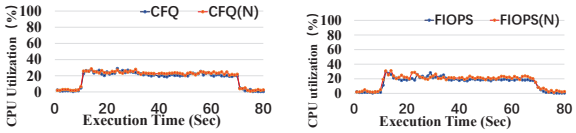
Figure 13(a) shows FV under CFQ and CFQ(N). *OLTP* almost occupies the whole device time when it is running alone. From 12s to 20s, *Sort* and *OLTP* are concurrently



(a) The changes of FV

(b) The changes of DU

Fig. 13: The changes of FV and DU when *OLTP*, *Sort*, and *Wordcount* are concurrently running



(a) CPU utilization for CFQ (b) CPU utilization for FIOPS

Fig. 14: CPU utilization comparison for CFQ and FIOPS

running. CFQ shows low FV, because *Sort*, which has high intensity and big request sizes, is aggressive. *OLTP* gets less than 10% device time. From 20s to 34s, three workloads are running together. CFQ shows higher FV than before, because *Wordcount* prevents *Sort* from occupying too many I/O resources. From 35s to 39s, *Wordcount* and *OLTP* are concurrently running. CFQ shows high FV, because *Wordcount* has low intensity. It will not occupy too many resources of *OLTP*. As for CFQ(N), FV is always above 0.9, because *NASS* can limit the request dispatch of *Sort*, and *OLTP* can immediately issue requests to SSD.

Figure 13(b) shows results of DU. CFQ(N) shows 10% improvement compared with CFQ when workloads are concurrently running, because *NASS* can prevent idleness caused by anticipation. Moreover, DU drops when I/O bursts end under CFQ, because anticipation makes resource owned by *Sort* or *Wordcount* cannot be released in time.

C. Overhead of NASS

NASS is integrated into the I/O scheduler layer. Hence, *NASS* does not add extra I/O latency. However, *NASS* evaluates aggressiveness, sequentiality, and deceptive idleness based on information of the historical requests. This means *NASS* may introduce CPU overhead. We run Filebench test (Section IV-B1) to compare the CPU utilization under CFQ(N) and FIOPS(N) to the CPU utilization under the original CFQ and FIOPS, respectively. While CFQ is the most widely used I/O scheduler among the four I/O schedulers, FIOPS is the most simple one and thus the overhead caused by NASS is most obvious in FIOPS.

Figure 14 shows CPU utilization under CFQ and FIOPS. Benchmarks start at 10s and end at 70s. The average CPU utilization under CFQ and CFQ(N) are 22.3% and 23.8%, respectively. The average CPU utilization under FIOPS and FIOPS(N) are 19.9% and 21.8%, respectively. This means that NASS only introduces 1.5% and 1.9% extra CPU overheads to CFQ(N) and FIOPS(N), respectively.

V. Related Works

Researchers have studied various solutions to improve I/O fairness in SSDs. Some research efforts have focused

on optimizations in the hardware layer (e.g., *Flash Translation Layer* (FTL) [3], *garbage collection* [1], and *cache management* [4]). For example, Chang *et al.* [3] propose a FTL algorithm which avoids interferences among users by mapping requests from different users into different channels to improve fairness. Huang *et al.* [5] propose to utilize the new hardware feature of SSDs (i.e., Open Channel SSD) to achieve fairness by directly exposing channels to applications. Given that hardware modifications is not practical, especially for the widely used SATA-based SSD, several works proposed solutions in the software layer, mainly by introducing SSD-friendly I/O schedulers. For example, FlashFQ [11] achieves fairness by leveraging anticipation. While CFFQ [15] abandons anticipation and uses simplified request queue to dispatch requests to guarantee fairness when I/O patterns of concurrent workloads are similar.

NCQ is the junction of the system layer and storage device. Chen *et al.* [16] show that parallelism of SSD can be better utilized by enabling NCQ to make SSD accept multiple I/O requests. Lee *et al.* [36] show that increasing the NCQ length can improve the transaction throughput of an OLTP system. Jaeho *et al.* [37] observe that NCQ on SSD causes frequent scheduling switch between queues of CFQ. This frequent switch hurts fairness. In contrast to related work, we introduce the first study that analyzes and shows how NCQ affects I/O fairness and SSD utilization. Moreover, we introduce NASS, a novel scheme that targets high fairness and device utilization by controlling the request dispatch of workloads based on their features.

VI. Conclusions and Future Works

A common practice to exploit parallelism in conventional SSDs is to use *native command queueing* (NCQ). In this study, we highlight, for the first time, the impact of NCQ on I/O fairness and SSD utilization. Therefore, we show by means of experimental evaluation how ignoring NCQ harms fairness and results in low device utilization. Guided by our analysis and observations, we propose a NCQ-aware I/O scheduling scheme (*NASS*). The key idea behind *NASS* is to elaborately control the request dispatch of workloads based on important features of the workloads to offer fairness and high device utilization. *NASS* is applied on the top of four state-of-the-art I/O schedulers: CFQ, BFQ, FlashFQ, and FIOPS. Experimental results demonstrate the effectiveness of *NASS* in offering better fairness and higher utilization of SSDs. In the future, we plan to analyze how SQ of NVMe affects I/O fairness and device utilization: NVMe requires more elaborate scheduler as it contains at most 64K SQs and each SQ can hold up to 64K requests.

Acknowledgements

This work is supported by National Key Research and Development Program under grant 2016YFB1000501, National Science Foundation of China under grant No.61732010, and Pre-research Project of Beifang under grant FFZ-1601. Shadi's work is partially funded by the Stack/Apollo connect talent project and the ANR KerStream project (ANR-16-CE25-0014-01).

References

[1] J. Kim, D. Lee, and S. H. Noh, "Towards SLO Complying SSDs through OPS isolation," *in FAST'15*, pp. 183–189.
[2] C. Li, D. Feng, Y. Hua, and F. Wang, "Improving RAID Performance Using an Endurable SSD Cache," *in ICPP'16*, pp. 396–405.
[3] D. W. Chang, H. H. Chen, and W. J. Su, "VSSD: Performance Isolation in a Solid-State Drive," *TODAES*, vol. 20, no. 4, pp. 51–64, 2015.
[4] S. Wu, Y. Lin, B. Mao, and H. Jiang, "GCaR: Garbage Collection aware Cache Management with Improved Performance for Flash-based SSDs," *in ICS'16*, pp. 1–12.
[5] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "FlashBox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs," *in FAST'17*, pp. 375–390.
[6] S. Lin, S. Lin, Y. Wang, Y. Wang, Y. Wang, and Y. Wang, "SDF: Software-Defined Flash for Web-Scale Internet Storage Systems," *in ASPLOS'14*, pp. 471–484.
[7] X. Jimenez, D. Novo, and P. Ienne, "Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance," *in FAST'14*, pp. 47–59.
[8] D. He, F. Wang, H. Jiang, D. Feng, J. N. Liu, W. Tong, and Z. Zhang, "Improving Hybrid FTL by Fully Exploiting Internal SSD Parallelism with Virtual Blocks," *TACO*, vol. 11, no. 4, pp. 1–19, 2014.
[9] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, and O. Mutlu, "FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives," *in ISCA'18*, pp. 397–411.
[10] *SSD Information Service CQ1 '18 Quarterly Update*, Trendfocus, 2018, https://www.digiliant.com/the-leader-in-ssd-shipments-2017-samsung-pid-36.html.
[11] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs," *in ATC'13*, pp. 67–78.
[12] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," *in FAST'12*, pp. 13–13.
[13] J. Guo, Y. Hu, B. Mao, and S. Wu, "Parallelism and Garbage Collection Aware I/O Scheduler with Improved SSD Performance," *in IPDPS'17*, pp. 1184–1193.
[14] *FIOPS*, Phoronix, 2018, http://www.phoronix.com/scan.php?page=news_item&px=MTAzOTU.
[15] M. Yi, M. Lee, and Y. I. Eom, "CFFQ: I/O Scheduler for Providing Fairness and High Performance in SSD Devices," *in IMCOM'17*, pp. 87–92.
[16] F. Chen, R. Lee, and X. Zhang, "Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-Speed Data Processing," *in HPCA'11*, pp. 266–277.
[17] Y. J. Yu, I. S. Dong, H. Eom, and H. Y. Yeom, "NCQ vs. I/O Scheduler: Preventing Unexpected Misbehaviors," *TOS*, vol. 6, no. 1, pp. 658–673, 2010.
[18] J. Axboe, "Linux block IO present and future," pp. 51–61, 2004.
[19] *BFQ*, Paolo, 2018, http://algo.ing.unimo.it/people/paolo/disk_sched/.
[20] B. Dees, "Native Cmmand Queuing - Advanced Performance in Desktop Storage," *Potentials*, vol. 24, no. 4, pp. 4–7, 2005.
[21] W. Jin, J. S. Chase, and J. Kaur, "Interposed Proportional Sharing for A Storage Service Utility," *in SIGMETRICS'04*, vol. 32, no. 1, pp. 37–48.
[22] *FIO*, Freecode, 2018, http://freecode.com/projects/fio.
[23] R. Jain, D. Chiu, and W. Hawe, "A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems," *in DEC'12*, vol. cs.ni/9809099.
[24] Y. Xu, A. Suarez, and M. Zhao, "IBIS: Interposed Big-Data I/O Scheduler," *in HPDC'13*, pp. 109–110.
[25] H. Tan, C. Li, Z. He, K. Li, and H. Kai, "VMCD: A Virtual Multi-Channel Disk I/O Scheduling Method for Virtual Machines," *TSC*, vol. 9, no. 6, pp. 982–995, 2016.
[26] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic Scheduling in Multi-Resource Clusters," *in OSDI'16*, pp. 65–78.
[27] *SSD*, Utilization, 2018, https://coderwall.com/p/utc42q/understanding-iostat.
[28] *The NASS project: Experimental-Results*, https://github.com/hao219/Experimental-results.
[29] L. M. Grupp, J. D. Davis, and S. Swanson, "The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs," *in FAST'13*, pp. 79–90.
[30] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. T. Kandemir, C. R. Das, and M. Jung, "Exploiting Intra-Request Slack to Improve SSD Performance," *in ASPLOS'17*, pp. 375–388.
[31] X. Ling, S. Ibrahim, S. Wu, and H. Jin, "Spatial Locality Aware Disk Scheduling in Virtualized Environment," *TPDS*, vol. 26, no. 9, pp. 2571–2585, 2015.
[32] X. Ling, S. Ibrahim, H. Jin, S. Wu, and S. Tao, "Exploiting spatial locality to improve disk efficiency in virtualized environments," *in MASCOTS'13*, pp. 192–201.
[33] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *in HotCloud'10*, pp. 10–16.
[34] *HiBench Big Data microbenchmark suite*, github, 2018, https://github.com/intel-hadoop/HiBench.
[35] R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," *in FAST'16*, pp. 1–10.
[36] S. W. Lee, B. Moon, and C. Park, "Advances in Flash Memory SSD Technology for Enterprise Database Applications," *in SIGMOD'09*, pp. 863–870.
[37] J. Kim, E. Lee, and S. H. Noh, "I/O Scheduling Schemes for Better I/O Proportionality on Flash-Based SSDs," *in MASCOTS'16*, pp. 221–230.