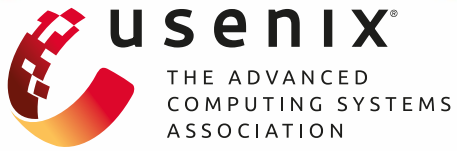


D2FQ: Device-Direct Fair Queueing for NVMe SSDs

Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong, *Sungkyunkwan University*

https://www.usenix.org/conference/fast21/presentation/woo

This paper is included in the Proceedings of the
19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings
of the 19th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

D2FQ: Device-Direct Fair Queueing for NVMe SSDs

Jiwon Woo, Minwoo Ahn, Gyusun Lee, Jinkyu Jeong
Sungkyunkwan University
{jiwon.woo, minwoo.ahn, gyusun.lee}@csi.skku.edu, jinkyu@skku.edu

Abstract

With modern high-performance SSDs that can handle parallel I/O requests from multiple tenants, fair sharing of block I/O is an essential requirement for performance isolation. Typical block I/O schedulers take three steps (submit-arbitrate-dispatch) to transfer an I/O request to a device, and the three steps incur high overheads in terms of CPU utilization, scalability and block I/O performance. This motivates us to offload the I/O scheduling function to a device. If so, the three steps can be reduced to one step (submit=dispatch), thereby saving CPU cycles and improving the I/O performance.

To this end, we propose D2FQ, a fair-queueing I/O scheduler that exploits the NVMe weighted round-robin (WRR) arbitration, a device-side I/O scheduling feature. D2FQ abstracts the three classes of command queues in WRR as three queues with different I/O processing speeds. Then, for every I/O submission D2FQ selects and dispatches an I/O request to one of three queues immediately while satisfying fairness. This avoids time-consuming I/O scheduling operations, thereby saving CPU cycles and improving the block I/O performance. The prototype is implemented in the Linux kernel and evaluated with various workloads. With synthetic workloads, D2FQ provides fairness while saving CPU cycles by up to 45% as compared to MQFQ, a state-of-the-art fair queueing I/O scheduler.

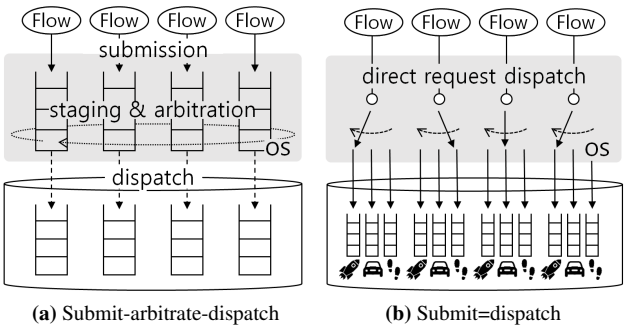
1 Introduction

Modern high-performance solid-state drives (SSDs) can deliver one million I/O operations per second (e.g., Samsung 980 Pro [1]). Such SSDs are also equipped with multiple I/O command queues to enable parallel I/O processing on multi-core processors. Thus, SSDs can accommodate multiple independent I/O flows in multi-tenant computing environments such as cloud data centers. In such an environment, fair sharing of the SSD performance is important to provide performance isolation between multiple applications or tenants.

A fair-share I/O scheduler [3, 8, 9, 29, 30, 34, 35] distributes storage performance proportionally to the weight of the applications. And, it is usually implemented at the block layer of the I/O stack. The typical block I/O scheduler takes three steps (submit-arbitrate-dispatch) during I/O processing (Figure 1a). When applications submit I/O requests, the I/O scheduling layer arbitrates and stages the I/O requests in the layer. Whenever an I/O scheduling condition is met (e.g., fairness), some staged I/O requests are eventually dispatched to the storage device. A problem is that these three-stage operations incur high CPU overhead, long I/O latency, and low I/O performance on high-performance SSDs. Since modern high-performance SSDs are shifting the bottleneck from I/O to CPU, many applications are changing their algorithms and/or data structures to adapt to the bottleneck changes [10, 17, 21]. With considering these efforts in reducing CPU overheads, reducing the CPU overheads associated with block I/O scheduling is also an important issue.

Offloading the I/O scheduling function to a device is an attractive approach to reducing the CPU overhead while preserving fairness. This scheduling offloading is already widely used in the domain of network packet scheduling [7, 23, 31, 32] since many network interface cards have device-side I/O scheduling features, such as round-robin scheduling. Fortunately, modern storage devices are now having a device-side I/O scheduling feature called NVMe weighted round-robin (WRR) queue arbitration. It provides three priority classes of I/O command queues, each with a configurable weight, and applies the weighted round-robin queue arbitration during I/O processing by the SSD firmware. However, a challenge is that the basic NVMe WRR is too simple to properly schedule I/O requests from multiple tenants with various I/O characteristics, such as a varied number of threads, different I/O request rates, and various request sizes.

This paper proposes D2FQ, a **d**evice-**d**irect **f**air **q**ueueing scheme for NVMe SSDs. D2FQ leverages the NVMe WRR feature but does not use it as it is. It abstracts the three queue classes as three-class queues with different I/O processing speeds. Then, for every I/O request submission, D2FQ selects an I/O command queue and dispatches an I/O request to the queue immediately (Figure 1b). The queue selection policy



(a) Submit-arbitrate-dispatch **(b)** Submit=dispatch

Figure 1: Typical I/O scheduling in the block layer (a) and the proposed I/O scheduling (b).

is carefully designed to provide fairness while reducing tail latency as much as possible. Since the arbitration step is removed and the submission and dispatch steps are unified in the block layer, D2FQ can minimize the CPU overhead during I/O scheduling and improve the I/O performance.

D2FQ also leverages a scalable yet sloppy minimum value tracking method. Similar to other fair-share I/O schedulers, D2FQ is virtual time-based. In these schedulers, it is important to track the minimum virtual time in a scalable way [11]. The proposed minimum tracking method tracks the minimum value almost always while having a small window of tracking a non-minimum value. However, this little possibility of incorrect tracking allows us to achieve scalability a lot as compared to a scalable minimum tracking object in the literature. [22]

D2FQ is implemented in the Linux kernel and evaluated with various workloads. Using the FIO benchmark with various workload configurations, our scheme provides fairness while reducing CPU utilization by up to 45% as compared to MQFQ [11], the state-of-the-art fair queueing scheme. When the storage device is not the bottleneck, D2FQ outperforms other schedulers in terms of I/O latency, CPU utilization, and reaches the maximum storage bandwidth faster than the other I/O schedulers. Since D2FQ unifies the I/O submission and dispatch steps into one, it can be integrated with the low-latency I/O stack, which has no I/O scheduling capability [19]. After the integration with the low-latency I/O stack, it outperforms other schemes, reducing the I/O latency by up to 35% and improving the I/O bandwidth by up to 54%.

This paper has the following contributions:

- We successfully demonstrate to build a fair-queueing I/O scheduler (D2FQ) on top of a simple yet efficient device-side scheduling feature (NVMe WRR). The only necessary abstraction is the device-side I/O queues with different I/O processing speeds.
- We propose a scalable yet sloppy minimum tracking method suitable for the virtual time-based fairness of D2FQ.
- We provide a detailed evaluation of the proposed fair-queuing I/O scheduler. The evaluation results demonstrate that D2FQ provides fairness, low CPU utilization, and high block I/O performance.

2 Background & Motivation

2.1 Fair Queueing for SSDs

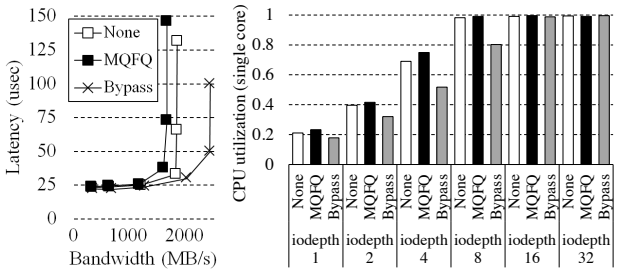
Modern high-performance SSDs are capable of accommodating parallel I/O requests from multiple tenants. For example, Samsung 980 Pro can perform at a million I/O operations per second [1]. This huge increase in the bandwidth and capacity of SSDs enable to service I/O requests from multiple independent workloads (or tenants) in a single storage device. Naturally, fair sharing of the SSD bandwidth is important to meet the service-level agreements of applications and to provide performance isolation between tenants. Among many proportional share I/O schedulers [3,8,9,29,30,34,35], virtual time-based fair queueing is an attractive solution for SSDs due to its work-conserving nature. They can maximize the SSD throughput while the bandwidth achieved by each tenant is proportional to the weight of the tenants.

An I/O flow is a stream of I/O requests issued by a resource principal [11] (e.g., virtual machines, Linux cgroups, thread groups), and the virtual time of a flow is the normalized accumulated I/O size serviced to the tenant. When a flow f is serviced an I/O request of length l, the virtual time vt_f of the flow is advanced by l/w_f where w_f is the weight of the flow.

Virtual time-based fair queueing I/O schedulers [9, 29] schedule I/O requests while minimizing the difference in virtual time between any flows. If all flows have the same virtual time, their I/O resource usages are proportional to their weights, and hence the fairness is satisfied. Accordingly, the goal of the schedulers is to minimize the virtual time gap between any flows. Hence, a flow with the minimum virtual time is only allowed to dispatch its I/O request since it is the flow with the lowest amount of I/O serviced. I/O requests from other flows are throttled by staging them in the I/O scheduler. This arbitration of request dispatching is denoted as I/O scheduling in the block layer, as shown in Figure 1a.

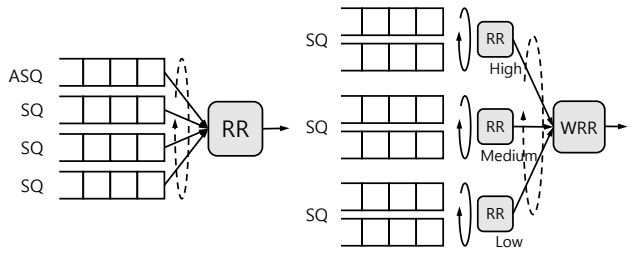
To maximize the performance of modern SSDs with internal parallelism, it is necessary to sustain a high number of in-flight requests. Accordingly, modern fair queueing I/O schedulers [11, 13] relax the strict fairness. Hence, flows whose virtual time is nearby the minimum virtual time are allowed to dispatch their I/O requests. This may allow a small amount of short-term unfairness but improves the overall I/O throughput by maximally utilizing the storage device.

Among fair queueing I/O schedulers, multi-queue fair queueing (MQFQ) [11] is the state-of-the-art approach for modern high-performance multi-queue SSDs. SSDs now have multiple command queues to utilize the internal parallelism of SSDs effectively. To scale with multiple command queues, MQFQ has request queues for each core in the I/O scheduler and employs scalable arbitration between the per-core I/O request queues. It employs two scalable objects: Mindicator [22] for scalable tracking of the minimum virtual time and a token tree [11] for scalable communication across cores.



(a) Latency/bandwidth **(b)** CPU (single core) utilization

Figure 2: (a) I/O latency and bandwidth and (b) CPU (single core) utilization of a single-thread 4 KB random read workload with varying I/O depth on Machine A in Table 1.



(a) Round robin **(b)** Weighted round robin

Figure 3: Two NVMe queue arbitration policies: (a) round-robin and (b) weighted round-robin.

With these objects, MQFQ keeps the number of in-flight requests high to maximize the performance of multi-queue SSDs while not significantly violating the short-term fairness. The MQFQ is prototyped in the block layer of the operating system (OS) I/O stack.

2.2 I/O Scheduling Overheads in Software

The block layer in the OS I/O stack (e.g., *blk-mq* in Linux [5]) is the core of block I/O scheduling. The generic block layer provides merging, reordering, staging, and accounting of I/O requests. In addition, block I/O schedulers (e.g., BFQ [4], mq-deadline [25], kyber [18]) are implemented as a module of the block layer. The multi-queue block layer maintains I/O *request queues* to stage I/O requests for I/O scheduling. Without I/O scheduling, the submitted requests are immediately dispatched to I/O *command queues* of a storage device (e.g., NVMe submission queues). With I/O scheduling, the scheduler arbitrates dispatching of I/O requests by staging them inside the scheduler. If the scheduling condition satisfies after processing other requests, the staged I/O requests are finally dispatched to the device.

With high-performance SSDs, however, the block layer incurs overheads in terms of CPU cycles and I/O latency. Accordingly, many studies have proposed to bypass the block I/O layer to achieve low I/O latency [19, 38]. Figure 2 shows how the overhead of the block layer affects I/O latency, I/O bandwidth, and CPU utilization. We compared the vanilla Linux kernel without I/O scheduling (*None*), *MQFQ*, and the light-weight block layer (*Bypass*) [19], which bypasses the block layer and submits I/O requests directly to the device's command queues. MQFQ shows the highest I/O latency and lowest I/O bandwidth in Figure 2a because the CPU is saturated earlier than the other schemes. None shows moderate performance, and Bypass shows the lowest I/O latency, highest I/O bandwidth, and lowest CPU utilization; it delays the saturation point further than the other schemes because of its lowest CPU overhead of block I/O service.

The high CPU cost of the block I/O scheduling can exacerbate the problem of CPU bottleneck in modern data-intensive applications with fast SSDs. With the introduction of low-latency SSDs, the performance bottleneck is moving from an I/O device to CPU [10, 17, 21]. This incurs the need for lowering the CPU contention by changing data structures and/or algorithms of applications. Hence, the CPU overheads caused by block I/O scheduling can also be addressed to fully harness the performance potential of high-performance SSDs today.

The high overhead of I/O scheduling can be alleviated by offloading I/O scheduling function to devices. Network interface cards (NICs) have experienced the era of microsecond-scale I/O latency earlier than SSDs. Many approaches have proposed to offload packet scheduling to NIC and succeeded in lowering the CPU utilization [7, 31, 32, 36]. Similarly, the block I/O scheduling can be offloaded to SSDs having device-side I/O scheduling features [14, 15, 26, 27, 33], and therefore the cost of the block I/O scheduling can be reduced.

2.3 Weighted round-robin in NVMe Protocol

Non-volatile memory express (NVMe) [26] is the de-facto standard interface bridging computer systems with storage devices due to its simplicity, efficiency, and scalability. The protocol also has a block I/O scheduling feature called weighted round-robin (WRR) queue arbitration [26]. The default I/O command scheduling policy of the protocol is round-robin; hence processing I/O commands one by one across command queues as shown in Figure 3a. If the WRR feature is enabled, the SSD firmware fetches I/O commands in a weighed round-robin fashion as shown in Figure 3b. With WRR enabled, command queues are classified into three priority classes (low, medium, and high)[1], and queues in each priority class are assigned a *queue weight* (1 – 256); hence queues in the same priority class share a queue weight. With WRR enabled, if queue weights are 1, 2, and 3 for the low, medium, and high queues, respectively, the SSD controller fetches three I/O commands from the high queues, then fetches two commands from the medium queues and then fetches one from the

[1]The NVMe WRR also supports another queue priority class called urgent priority but our scheme does not consider the use of the class

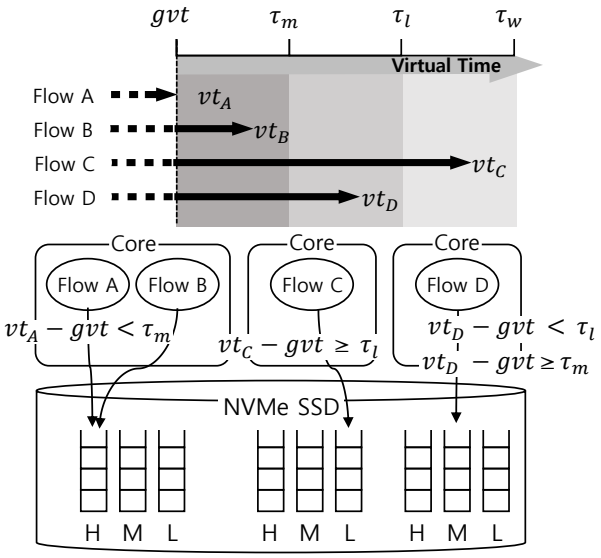


Figure 4: Overview of D2FQ.

low queues. Queues in the same priority class are accessed in a round-robin fashion.

The NVMe WRR feature can be easily implemented inside the SSD because of its simplicity. However, applying this to fair queueing has many challenges to be resolved. First, NVMe WRR has only three priority classes (i.e., low, medium and high), whereas the number of tenants can be higher. Second, its queue arbitration does not consider I/O sizes. Finally, the weight ratio between any two queues could not directly match the ratio of I/O commands serviced from the two queues. This is because the number of I/Os actually processed can vary depending on the utilization of the command queues. Consequently, it is necessary to bridge the gap between the requirement of fair queueing and the simple yet uncertain performance characteristic of NVMe WRR.

3 Device-Direct Fair Queuing

3.1 Overview

This paper proposes a fair queueing scheme called device-direct fair queueing (D2FQ) for NVMe SSDs. D2FQ offloads the I/O scheduling functionality to an SSD by exploiting the NVMe WRR feature. Accordingly, the CPU overheads and I/O latency associated software I/O scheduling can be reduced. Figure 4 shows the overview of D2FQ.

D2FQ is a virtual time-based fair queueing scheme. It manages the virtual time of each flow and the global virtual time (*gvt*), the minimum virtual time among active flows. An *active* flow is a flow with any pending I/O requests to be served. As other fair queueing schemes do, D2FQ provides fairness between only active flows.

D2FQ throttles a flow if its virtual time is far ahead of *gvt*. Throttling is done not by the block layer of the I/O stack but by exploiting the NVMe WRR feature. In addition, D2FQ does not establish any fixed mapping between flows and I/O command queues. Instead, our scheme abstracts the three classes of queues as three different queues with different I/O processing speeds (fast, moderate and slow). Then, whenever a flow submits an I/O request, our scheduling policy immediately selects a queue of the desired speed and dispatch the request to the queue (Figure 1b). As a result, slow flows in the virtual time domain are enforced to use the fast queues to catch up the virtual time of other flows, and fast flows are throttled by using the slow or moderate queues.

D2FQ maintains three threshold values: τ_m, τ_l and τ_w; the former two thresholds are used during the queue selection, and the latter is used to detect unfairness which is explained later in Section 3.2. When a flow f issues an I/O request, the gap between its virtual time and *gvt* (i.e., $vt_f - gvt$) is compared with the two threshold values to select the class of command queue (SQ) for I/O dispatching as follows:

$$SQ = \begin{cases} Q_{high} & \text{if } vt_f - gvt < \tau_m \\ Q_{mid} & \text{else if } vt_f - gvt < \tau_l \\ Q_{low} & \text{otherwise} \end{cases} \quad (1)$$

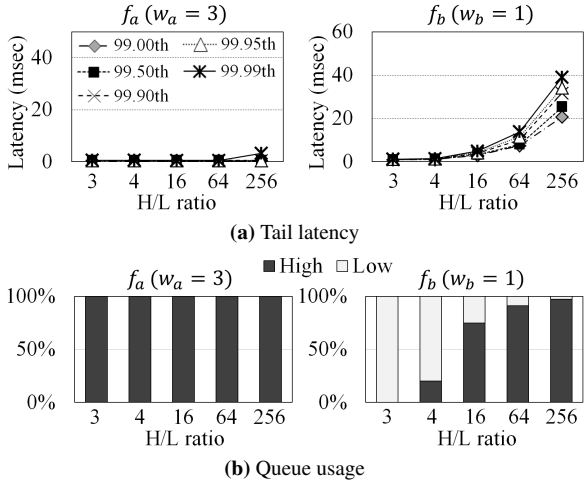
Hence, if the virtual time of a flow is not far from *gvt* (vt_B in Figure 4), its I/O requests are queued to high queues; hence the flow is not throttled. If a flow is far ahead of *gvt* (vt_C in the figure), its I/O requests are queued to low queues; hence the flow is throttled. Please note that our scheme assumes all cores are having their own queue set (three queues of each priority class).

Example walkthrough. Let us assume two flows f_a and f_b and their weights $w_a = 3$ and $w_b = 1$. Both flows issue 4 KB I/O requests with high I/O depth. If the weight of high queues is 3 and the weight of low queues is 1, both flows can fairly share the bandwidth by making flow f_a use the high queues and flow f_b use the low queues. However, our scheme does not statically map any flow to any queue but establishes the mapping dynamically. Indeed, at the beginning, both flows use the high queues together because their virtual time gap is zero. Then, vt_a advances by $4\,KB/3$ while vt_b advances by $4\,KB/1$ on each I/O completion; consequently, vt_b advances 3 times faster than vt_a. In the end, $vt_b - gvt(= vt_a)$ exceeds τ_l, and flow f_b begins to use the low queues. After that, both flows have the same virtual time progress rate.

We define the term *H/L ratio* as the ratio of the weight of high queues over the weight of low queues. The H/L ratio is the most important factor in satisfying the I/O fairness. It determines the maximum speed difference the high and low queues can produce if I/O sizes are identical and the queues are fully utilized. Hence, it determines the maximum weight ratio our scheme can cover with fairness.

A small H/L ratio cannot meet the fairness requirement. In the previous example, if the H/L ratio is 2, the two flows f_a and f_b cannot fairly share the I/O bandwidth.

Meanwhile, a high H/L ratio has a wide coverage of weight



(a) Tail latency

(b) Queue usage

Figure 5: Effect of the H/L ratio to tail latency and queue usage.

ratios between any flows. In the above example, if the H/L ratio is 6, three fifth of I/O requests from f_b need to use the high queues, and then the two flows can meet the fair bandwidth distribution. Since the low queues are $1/6$ times slower than the high queues, which is more than necessary, f_b needs to use the high queues to compensate the penalty caused by the use of the low queues.

The use of high H/L ratio seems appropriate. However, this has a side-effect of increasing the tail of I/O latency. Figure 5 shows the tail latency and queue utilization of flow f_a and f_b with varying the H/L ratio from 3 to 256. As shown in the figure, the H/L ratio of 3 shows the lowest tail latency for f_b. In that configuration, f_b uses the low queues only while the f_a uses high queues only. However, with high H/L ratios, flow f_b shows high tail latency while increasing its usage portion of the high queues. The flow f_b needs to be throttled but the use of the low queues with high H/L ratio gives higher penalty than necessary. This results in the increase of tail latency and the increase of the high queue usage.

Although the above examples show a simple workload having only two flows with a fixed I/O size and high I/O submission rate. However, real-world workloads may have a various number of flows with any number of threads, I/O submission rates and I/O sizes. With these realistic and unknown I/O characteristics, it is challenging to find the proper H/L ratio to make queues with sufficient I/O processing speed difference.

3.2 Dynamic H/L Ratio Adjustment

D2FQ finds proper weights of the three queue classes to meet the two goals: providing fairness and taming tail latency. As explained above, the H/L ratio is the most important factor since D2FQ needs to satisfy fairness. In this regard, our scheme finds a proper H/L ratio first and then sets the weight of medium queue as the square root of the H/L ratio. Hence,

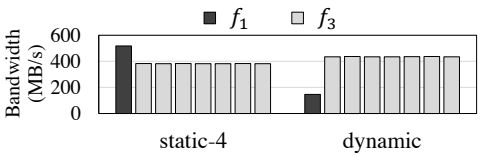


Figure 6: Effect of the dynamic H/L ratio adjustment.

the speed ratio between high and medium queues is equal to that between medium and low queues.

D2FQ collects information of the virtual time of all flows and their I/O usage statistics, such as queues and I/O sizes. It uses the collected information to find the appropriate H/L ratio periodically as follows.

3.2.1 Increasing H/L Ratio

The H/L ratio needs to increase when fairness is not satisfied. Recall that our scheme maintains three thresholds, and the third one τ_w is the threshold to detect unfairness and trigger the process of finding a proper H/L ratio. Hence, if a flow with the largest virtual time is a far ahead of *gvt* by τ_w, D2FQ finds a new H/L ratio that is suitable in providing fairness.

To this end, D2FQ keeps track of two flows, one with the largest virtual time (denoted as f_{max}) and the other with the smallest virtual time (denoted as f_{min} whose virtual time $vt_{f_{min}}$ is equal to *gvt*). Then, it calculates the delta of virtual time increase in the last information collection period. Hence, Δvt_{max} is the virtual time increase rate in the last period by f_{max}, and Δvt_{min} is the virtual time increase rate last period by f_{min}. Then, the next H/L ratio is calculated by using the following formula:

$$\text{H/L ratio}_{next} = \lfloor \frac{\Delta vt_{max}}{\Delta vt_{min}} \times \text{H/L ratio}_{prev} \rfloor + 1 \qquad (2)$$

The term $\frac{\Delta vt_{max}}{\Delta vt_{min}}$ is the ratio of widening virtual time gap between f_{max} and f_{min}, and this has happened under the previous H/L ratio. Accordingly, the next H/L ratio should be the product of the widening ratio and the previous H/L ratio. The next H/L ratio is ensured to have a higher value by one than the proper H/L ratio to make the gap narrowed down next.

The use of high queues does not guarantee that I/O requests in the high queues are processed faster than those in the low queues. However, our dynamic weight adjustment finds out a proper H/L ratio to meet the fairness. Figure 6 shows the bandwidth distribution of eight flows, one with weight 1 (f_1) the other seven flows with weight 3 (f_3). When the H/L ratio is fixed to 4 (*static-4*), f_1 uses the low queues and seven f_3 flows use the high queues. In this case, all the flows are not allocated fair amount of I/O resource due to the contention in the high queues. However, if our *dynamic* H/L ratio adjustment is applied, the H/L ratio becomes 22 using Equation 2 and all the flows meet the fair bandwidth distribution; the required effective queue weight ratio is 1:21 (1:3 weight ratio with 1:7 ratio of the number of flows) and one is incremented using the equation.

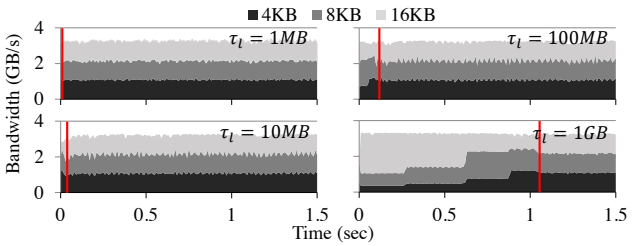


Figure 7: Time-series bandwidth of three flows with different I/O sizes (4 KB, 8 KB and 16 KB) with varying the threshold τ_l from 1 MB to 1 GB.

3.2.2 Decreasing H/L Ratio

As explained in Section 3.1, an unnecessarily high H/L ratio may increase the tail latency of flows requiring throttling. Hence, it is necessary to decrease the H/L ratio if the current H/L ratio is too high. The condition to decrease the H/L ratio is when the maximum virtual time gap is below τ_w. In this case, D2FQ calculates the *virtual slowdown* of each flow using the I/O statistics in the last statistics collection period. The virtual slowdown is an estimated value of how the I/O requests of this flow are slowed down by not using the high queues. The virtual slowdown of flow f is calculated by using the following formula where $\sum l_{f,x}$ is the total amount of I/O submitted to the queue class x by flow f in the last period and $\frac{p_x}{p_y}$ is the weight ratio of two queue classes x and y:

$$\text{slowdown}(f) = \frac{\sum l_{f,h} * \frac{p_h}{p_h} + \sum l_{f,m} * \frac{p_h}{p_m} + \sum l_{f,l} * \frac{p_h}{p_l}}{\sum l_{f,h} + \sum l_{f,m} + \sum l_{f,l}} \quad (3)$$

Then, D2FQ chooses the maximum virtual slowdown among all active flows in the system and sets the next H/L ratio as the maximum value.

3.3 Determining Thresholds

D2FQ regulates the fairness by throttling fast flows in virtual time (i.e., flows with low weight values). The two thresholds (τ_m and τ_l) are the criteria of when to throttle such fast flows.

Large threshold values allow a huge virtual time interval between any flows and *gvt*. Hence, it determines the allowed unfairness in virtual time.

Figure 7 shows the time-series bandwidth of three flows with three I/O sizes: 4 KB, 8 KB and 16 KB, respectively. The vertical line in each figure indicates the point in time starting fair bandwidth sharing. As shown in the figure, with a small threshold (τ_l = 1 MB[2]), the three flows equally share the I/O bandwidth from the beginning. That point is delayed to after 1 second with a large threshold value (1 GB). However, after that point, the flows equally share the I/O bandwidth.

[2] τ_l = 1 MB indicates that a flow with weight 8 can cross the threshold boundary after it is serviced 1 MB I/O size. If its weight is 1, the flow can meet the threshold only after 128 KB I/O size serviced (one eighth of 1 MB).

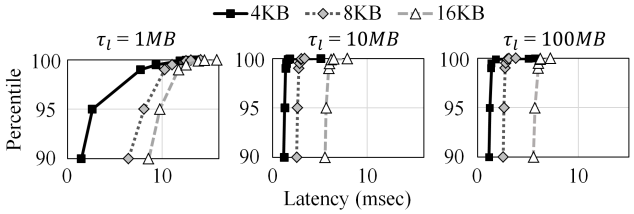


Figure 8: Tail latency of the three flows with varying the threshold τ_l: 1 MB, 10 MB and 100 MB.

In our scheme, the virtual time progression is controlled by making the flow crossing the threshold boundary back and forth. Thus, a flow could not get proper throttling until the flow hits thresholds in the virtual time domain. Hence, the threshold values only determine when this control begins.

On the other hand, small threshold values may unintentionally increase the tail latency of flows, especially those who stay back in the virtual time domain. Such flows are intended to use the high queues only. However, a small increase in virtual time can make such flows use the medium or low queues due to crossing the thresholds. This may exacerbate the tail latency of all flows because other flows can unintentionally use the high queues and be throttled to offset the benefits of using the high queues.

Figure 8 compares the tail latency of the three flows with varying threshold values: 1 MB, 10 MB, and 100 MB. As shown in the figure, with the 10 or 100 MB threshold values, the three flows show no significant increase in the tail latency. However, with the 1 MB threshold value, the three flows show up to 3.7 times long tail latency.

Consequently, there is a trade-off between short-term fairness and tail latency in setting the threshold value. Depending on whether a user focuses on tail latency or short-term fairness, the user can adjust the appropriate threshold value.

The characteristics of workloads, especially I/O size and weight of flows, impact on the selection of the threshold values because the I/O size and weight determine the stride of virtual time increase. Our scheme uses a proper τ_l that is empirically found to work with our tested workloads. We leave the fine-tuning of the threshold values to the users or system administrators.

τ_m also affects tail latency. However, its latency impact is not significant as compared to that of τ_l since the medium queues are faster than the low queues. We empirically found that it is suitable to set $\tau_m = \tau_l/2$.

3.4 Global Virtual Time Tracking

The value *gvt* is frequently accessed during I/O submission and completion. Accordingly, it is important to track *gvt* in a scalable way.

Tracking *gvt* is equal to tracking the minimum among a set of values where each value changes simultaneously. One coarse-grained approach is to inspect all the values for every

```
1  struct vt {
2    u64 id : 16 bits // id of a flow
3    u64 vt : 48 bits // virtual time of a flow
4  } gvt; // global virtual time
5
6  void update_gvt (vt my)
7    while (true) {
8      vt old = gvt
9      if ((old.id == NO_HOLDER)
10     || (old.id == my.id && old.vt < my.vt)
11     || (old.id != my.id && old.vt > my.vt)) {
12       if ( CAS(&gvt, old, my) == SUCCESS )
13         return
14     } else
15       return
16   }
17
18 void release_gvt (vt my)
19   while (true) {
20     vt new, old = gvt
21     if (old.id == my.id) {
22       new.id = NO_HOLDER; new.vt = old.vt
23       if ( CAS(&gvt, old, new) == SUCCESS )
24         return
25     } else
26       return
27   }
```

Figure 9: Pseudocode of tracking the global virtual time.

query of the minimum. An alternative is to use a scalable minimum tracking object such as Mindicator [22].

In D2FQ, we take yet another approach of tracking *gvt* in a sloppy way. We consider that it is not always necessary to retrieve the true minimum value among the virtual time of flows. The goal of tracking *gvt* is not to minimize the virtual time gap between any flows. It is to make the pace of virtual time progression of any flows at a similar rate. If the value of *gvt* is not far from the true minimum value, the sloppy management hardly affects the policy of low or medium queue selection since the use of the thresholds gives tolerance to the queue selection policy.

In this regard, our scheme maintains the *gvt holder*, the flow owning *gvt*, and allows only the *gvt* holder to be able to increase *gvt* (line 9–13 in Figure 9). Other flows can also update *gvt* but only when their virtual time is smaller than *gvt* (line 11). A little inaccuracy can happen when the *gvt* holder increases *gvt* and it now overtakes the virtual time of other flows, hence violating that *gvt* is not the minimum. However, this little inaccuracy comes with the simplification of the *gvt* update operation; otherwise, every *gvt* update needs to inspect the virtual time of all the flows.

The function `update_gvt()` is called when I/O completion happens. When the *gvt* holder becomes inactive, it calls `release_gvt()`, and any flow can become the *gvt* holder. We use the atomic instruction `compare_and_swap` (`CAS`) and the while loop to secure minimal serialization between concurrent *gvt* updates.

3.5 Implementation

D2FQ is implemented in the multi-queue block layer [5] of the Linux kernel. Figure 10 represents the high-level pseudo

```
1  per-CPU structures:
2      high/medium/low class SQ
3
4  per-flow structures:
5      vt              // virtual time
6      nr_inflight     // # of in-flight requests
7      weight          // I/O weight of this flow
8
9  void dispatch_request (request R, flow F)
10     if (F->active == false)
11         F->active = true; F->vt = gvt
12     F->nr_inflight += 1
13     vt_gap = F->vt - gvt
14     if (vt_gap > threshold_low)
15         R->dispatch_Q = low class SQ
16     else if (vt_gap > threshold_medium)
17         R->dispatch_Q = medium class SQ
18     else
19         R->dispatch_Q = high class SQ
20
21 void complete_request (request R, flow F)
22     F->vt += R->length / F->weight
23     F->nr_inflight -= 1
24     if (F->nr_inflight == 0)
25         enter_grace_period(F)
26     update_gvt()
```

Figure 10: Pseudocode for D2FQ working flow.

code of D2FQ. Each core has three submission queues (high, medium and low) (line 2). Each flow has virtual time, the number of in-flight requests and its weight value (line 4–7).

The function `dispatch_request()` (line 9) is the core function that selects the queue to dispatch an I/O request. It is invoked in the block layer function `blk_mq_start_request()`. However, D2FQ is independent to the block layer since it has no staging operation. Accordingly, it can also be invoked elsewhere before request dispatching, such as `nvme_queue_rq()`.

The number of in-flight requests is used to detect the activeness of flows. If it becomes zero, a grace period is given, and after that the flow becomes idle (line 24). The use of the grace period is to avoid the deceptive idleness [12].

The function `complete_request()` is invoked whenever a request is completed. In our implementation, it is called from the block layer function `blk_mq_finish_request()`. It is also independent to the block layer so it can be invoked elsewhere after request completion.

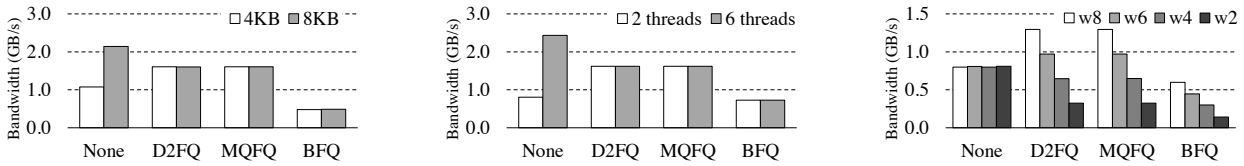
4 Evaluation

4.1 Methodology

Table 1 shows our experimental configuration. We used Samsung Z-SSD as the main storage device because it supports the NVMe WRR feature. The NVMf on ramdisk is used only for the scalability test due to lack of WRR support.

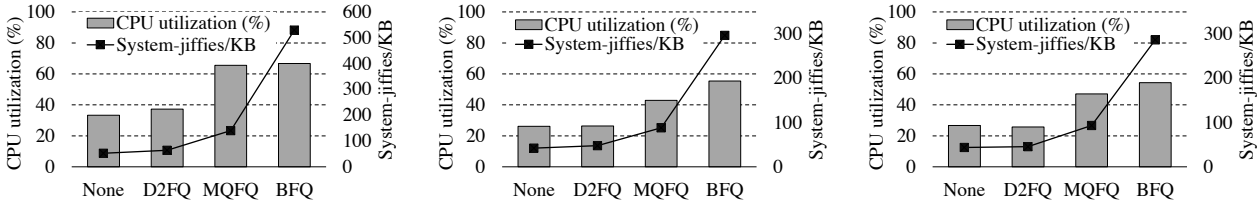
We evaluated the following four schedulers:

- **None** performs no I/O scheduling in the block layer.
- **D2FQ** is the prototype of our scheme which is based on None as explained in Section 3.5. The dynamic H/L ratio adjustment is enabled. The H/L ratio is initially 256. The



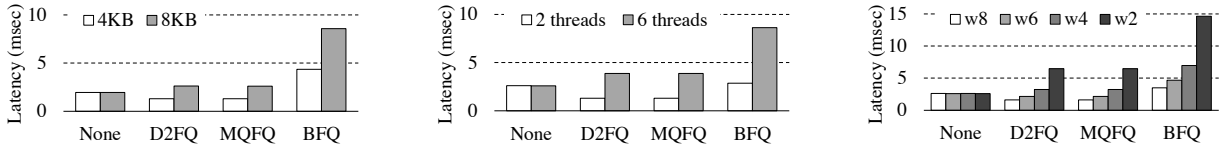
(a) I/O request size (4 KB vs 8 KB) **(b)** # of threads (2 vs 6) **(c)** Weight ratio 8:6:4:2

Figure 11: Fairness measurement with varying (a) I/O request size, (b) the number of threads and (c) the weight of each flow.



(a) I/O request size (4 KB vs 8 KB) **(b)** # of threads (2 vs 6) **(c)** Weight ratio 8:6:4:2

Figure 12: CPU utilization and I/O processing cost of the workloads in Figure 11.



(a) I/O request size (4 KB vs 8 KB) **(b)** # of threads (2 vs 6) **(c)** Weight ratio 8:6:4:2

Figure 13: Average I/O latency of the workloads in Figure 11.

Hardware configuration	**Machine A**	
	CPU	Intel Xeon Gold 5112 3.60 GHz 8 cores
	Memory	DDR4 192 GB
	Storage	Samsung Z-SSD 800 GB
	Machine B	
	CPU	Intel Xeon Gold 6226 2.70 GHz 24 cores
	Memory	DDR4 192 GB
	Storage	Samsung Z-SSD 800 GB NVMf ramdisk 64 GB
	Network	Mellanox Connect X-4 56 Gbps
Software configuration	**OS**	Ubuntu 18.04.4
	Kernel	Linux version 5.3.10
	FIO	libaio, random read, direct I/O
	YCSB	Uniform request distribution

Table 1: Experimental configuration.

threshold τ_l is set to 8 MB for flows with weight 8 which is 1 MB for flows with weight 1; then the rest of the thresholds are automatically set $\tau_m = \tau_l/2$, $\tau_w = 2 \times \tau_l$. The period of the H/L ratio adjustment is set to one second.

- **MQFQ** is the state-of-the-art fair-queueing I/O scheduler. Unfortunately the source code is unavailable. So we made our own implementation of MQFQ, which faithfully follows the design described in the MQFQ paper [11]. We ported the Mindicator written in C++ [24] to C for the integration with the Linux kernel. MQFQ has two parameters: D is 64 and T is 45 KB in our setting.
- **BFQ** is the time slice-based proportional share I/O scheduler in Linux [4]. We set `max_budget` to 256.

The four schedulers use `ionice()` to set the weight of each flow. The weight values range from 1 to 8 in the experiment. Unless specified, the default weight value is 8.

The evaluation is organized to answer the three questions: (1) whether our scheme provides fairness when the storage device is saturated, (2) how well our scheduler shows good I/O performance when the storage device is unsaturated, and (3) whether our scheme provides fairness with realistic workload.

4.2 Fairness

Providing fairness is the primary goal of fair queueing when multiple flows contend on a storage device. We build three workloads with varying the following factors: I/O request size, the number of threads and the weight of flows. Then, we measure the bandwidth of each flow in Figure 11. Unless specified, the I/O depth is 128 by default in each thread.

I/O Request Size. Figure 11a shows the bandwidth distribution of two I/O flows with different I/O request sizes: 4 KB vs 8 KB. Each flow has 4 threads. Since the two flows have the same weight, the SSD bandwidth should be fairly distributed to the two flows, and D2FQ, MQFQ and BFQ provide the fairness. The total bandwidth is identical across the three schedulers: None, D2FQ and MQFQ.

Thread Count. Figure 11b shows the bandwidth distribution of two I/O flows with different number of threads: 2 threads vs 6 threads. The I/O request size is 8 KB in both flows. In this experiment, both flows need to evenly share the SSD bandwidth and D2FQ, MQFQ and BFQ achieve this while None does not.

Weight. Figure 11c shows the bandwidth consumed by four I/O flows with different weight values: 8, 6, 4 and 2; each

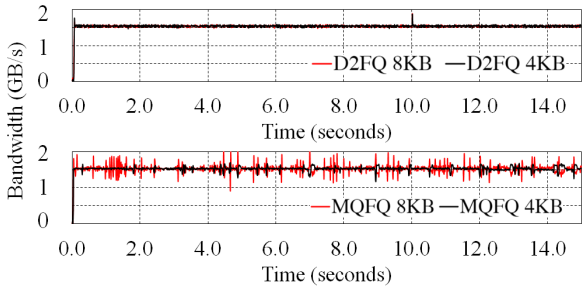


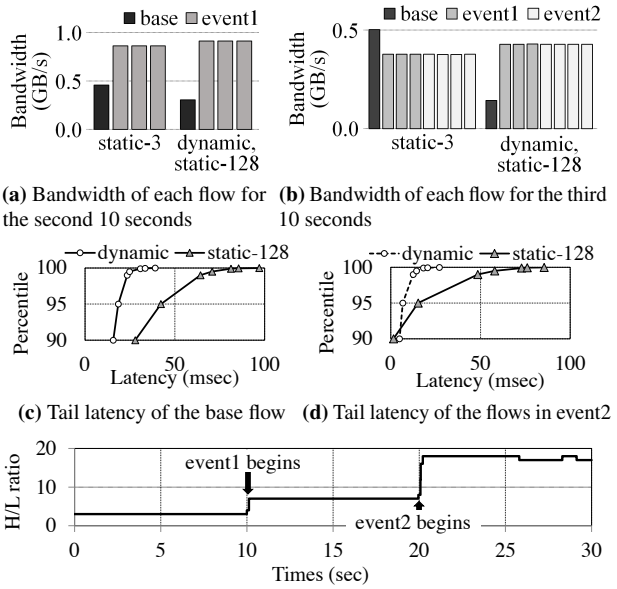
Figure 14: Bandwidth fluctuation test between fair I/O schedulers (same configuration as figure 11a).

flow has two threads, and the request size is 8 KB. Hence, the bandwidth ratio of the four flows needs to be identical to the ratio of weights. As shown in the figure, D2FQ and MQFQ show correct bandwidth distribution ratio while BFQ shows a slightly poor bandwidth distribution. In the three workloads, the total bandwidth of BFQ is lower than the others. Please note that, in the above three experiments with D2FQ, the H/L ratio is 256 at the beginning and converges to 3, 3 and 6, respectively, by our dynamic H/L ratio adjustment.

Cost of Fairness. The cost of I/O scheduling is CPU cycles and D2FQ minimizes them by avoiding arbitration including request staging in the block layer. This effect results in the reduction in CPU cycles or CPU overhead for each I/O request handling. Figure 12 shows the CPU utilization and system jiffies per 1 KB I/O of the three workloads in Figure 11. As shown in the figure, BFQ shows the highest CPU utilization and also fails to fully utilize the SSD. MQFQ shows similar CPU utilization to BFQ due to its computation for request arbitration. D2FQ consume slightly more CPU cycles than None since D2FQ needs a few CPU cycles to select queue during dispatch and maintain scheduler statistics, such as virtual time. D2FQ reduces the CPU utilization by up to 45% as compared to MQFQ in Figure 12a. If the metric is CPU cycles per I/O (i.e., system-jiffies/KB), None and D2FQ show lower per-request CPU cost than MQFQ and BFQ. In summary, D2FQ provides fairness with minimal CPU cycles and the saved CPU cycles may be able to be used more usefully for applications.

Latency. Figure 13 shows the average I/O latency of the three workloads in Figure 11. With the fair I/O schedulers, the I/O latency is largely affected by I/O throttling; the throttled flows show longer I/O latency than the others. D2FQ and MQFQ show similar latency results. However, BFQ shows longer latency than the two schedulers. None shows the shortest latency in all the cases but fails to provide the fairness.

Short-term Fairness. While other fair queueing schedulers [9, 11, 13] have theoretical upper-bound in unfairness, D2FQ has none. This leads us to measure how much short-term unfairness occurs at least empirically. To this end, we measure the bandwidth of each flow of the request size experiment (Figure 11a) every short time interval (10 ms) and depicts the time-series bandwidth in Figure 14. Interestingly,



(a) Bandwidth of each flow for the second 10 seconds **(b)** Bandwidth of each flow for the third 10 seconds

(c) Tail latency of the base flow **(d)** Tail latency of the flows in event2

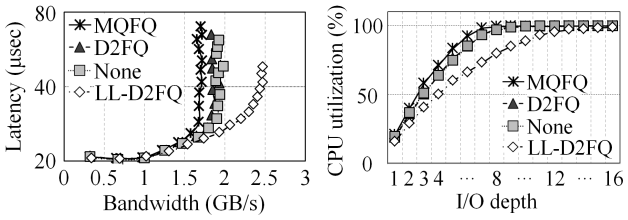
(e) Dynamic H/L ratio change over time

Figure 15: Effect of the dynamic H/L ratio adjustment.

our scheme shows very stable bandwidth distribution in both flows. This indicates that both flows almost evenly share the SSD bandwidth in every short time period. Meanwhile, MQFQ shows fluctuation of bandwidth in both flows; this phenomenon is due to frequent exchange of dispatch slots across the cores/sockets.

Dynamic H/L Ratio Adjustment. To test the effect of our dynamic H/L ratio adjustment, we build a synthetic workload with three flow groups with different lifetimes: (1) *base* that contains one weight-1 flow and runs from the beginning to the end, (2) *event1* that contains three weight-3 flows and runs from the 10-second point in time to the end, and (3) *event2* that contains four weight-3 flows and runs from the 20-second point to the end. Each flow has 2 threads and issues 4 KB I/O requests with 128 I/O queue depth. We run the workload with the three H/L ratio configurations: *static-3*, *static-128*, and *dynamic*, and depict the bandwidth, latency and H/L ratio change (dynamic only) in Figure 15.

Static-3 fails to provide the bandwidth fairness in Figure 15a and 15b; the bandwidth ratio between base and the rest should be 1:3. Static-128 and dynamic achieve the target bandwidth ratio because their H/L ratios are high enough to satisfy the fairness. However, as shown in Figure 15c and 15d, static-128 shows long tail latency because the H/L ratio of 128 is too high and flows using the low queues experience long I/O delays. Our dynamic H/L ratio adjustment adaptively sets the H/L ratio properly based on the I/O patterns of the flows. The H/L ratio changes over time as shown in Figure 15e; it is set to 7 when event1 begins and to 17–18 when event2 begins. Please note that the H/L ratio is not 22 as in Section 3.2.1 because the use of the medium queues gives additional fairness control over the low queues.



(a) Bandwidth and latency **(b)** Single-core utilization

Figure 16: Latency, bandwidth and CPU (single core) utilization with a single-thread flow with varying I/O depths.

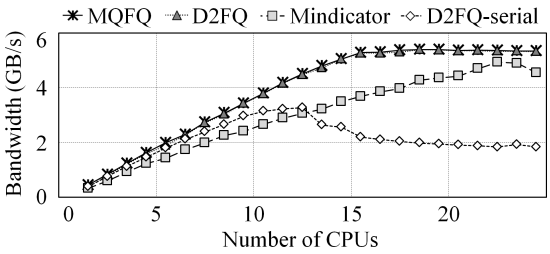


Figure 17: Scalability test of the global virtual time tracking schemes.

4.3 I/O Performance

The I/O scheduler performs not only when the storage device is saturated but also when the device is unsaturated. When the device is unsaturated, the performance of I/O scheduler is the major factor to the I/O performance. In this regard, we measure the I/O performance under low I/O contention.

In this experiment, we add another scheme to demonstrate that D2FQ is independent of the multi-queue block layer in Linux. A low-latency I/O stack [19] achieves low I/O latency by its submit=dispatch characteristic. Unfortunately, it lacks I/O scheduling support, which can be complemented by D2FQ; *LL-D2FQ* is the low-latency I/O stack with D2FQ, thereby having the I/O scheduling capability.

Figure 16 shows the latency, bandwidth and CPU utilization of each I/O scheduler. For the workload, we run a single-thread FIO performing 8 KB random read with varying its I/O depth from 1 to 16.

Basically, the increase in I/O depth results in the increase of bandwidth as well as latency in all the schedulers as shown in Figure 16a. Unless the CPU is saturated, increasing the I/O depth increases the CPU utilization due to handling more I/O requests as shown in Figure 16b.

More importantly, the overhead of I/O scheduler significantly affects the I/O latency, I/O bandwidth and the CPU utilization. As shown in the figure, MQFQ shows the lowest performance in terms of the three metrics. D2FQ and None show similar position in performance since they exclude the I/O scheduler in the block layer. Finally, LL-D2FQ outperforms the others by up to 35% in latency and 54% in bandwidth due to its low overhead in I/O request handling. It delays the CPU saturation point to the I/O depth of 14 whereas the other schedulers saturate the CPU at the I/O depth of 9 in Figure 16b.

Scalability. We test the scalability of D2FQ with varying the *gvt* tracking methodology. *None* performs no *gvt* tracking. *D2FQ* uses our *gvt* tracking method in Section 3.4. *Mindicator* tracks *gvt* using Mindicator [22] as in MQFQ [11]. *D2FQ-serial* tracks *gvt* by inspecting all flows every time of accessing *gvt*.

As shown in Figure 17, with increasing the number of cores (i.e., flows), D2FQ-serial does not scale after 12 cores due to cross-socket communication [6]. Mindicator scales well, but its tree-based data management incurs overheads with high core counts. D2FQ shows identical scaling to None.

4.4 Realistic workload

Finally, we measure the impact of D2FQ on realistic workloads. We run two flows: one flow of YCSB workloads [37] on the RocksDB and the other contending flow of the FIO benchmark. Since we run a realistic workload, YCSB saturates CPU first in Machine A. So, we use Machine B in this experiment. This machine has another difficulty in queue assignment since 24 cores need 72 command queues (three queues for each core), but Z-SSD provides only 32 command queues. To resolve this issue, we group three queues of three adjacent cores and make the three cores share the three queues as the three-class queues in this experiment.

The FIO workload issues 4 KB random read using 4 threads with 128 I/O depth. In the YCSB workload performs 64 million operations on 64 GB key-value dataset with 1 KB value size. The physical memory is reduced to 16 GB.

Figure 18 shows the bandwidth and CPU utilization of the workload with the four schedulers. As shown in the figure, None cannot fairly distribute the SSD bandwidth to the two flows. YCSB consumes lower bandwidth than FIO since FIO is more bandwidth hungry. With fair queueing schedulers, the bandwidth is fairly distributed across the two flows; D2FQ shows 1.00 – 1.05 bandwidth ratio (YCSB bandwidth over FIO bandwidth) and MQFQ shows 1.00 – 1.08 bandwidth ratio. None shows lower CPU utilization than D2FQ and MQFQ because YCSB, which consumes more CPU cycles per I/O than FIO, takes a smaller portion in total bandwidth than D2FQ and MQFQ.

D2FQ shows a slightly higher total bandwidth than MQFQ by up to 1.83% on average. This is due to the true work-conserving characteristic of D2FQ; all submitted requests are dispatched to the device queues. MQFQ rarely fails to maximally utilize the device bandwidth due to the exchange of request dispatch slots between cores and sockets.

Figure 19 shows the average I/O latency of each workload normalized to the result of None. The YCSB workload reports the latency and number of operations for each operation type, so we calculate the weighted average latency in that case. As shown in the figure, basically flows showing higher I/O bandwidth achieve shorter I/O latency. With None, FIO

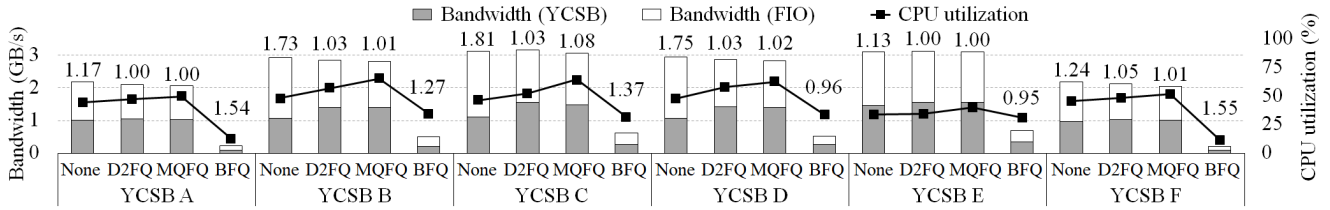


Figure 18: I/O bandwidth and CPU utilization of the realistic workload (YCSB with FIO). The number above each histogram shows the bandwidth ratio (the FIO bandwidth over the YCSB bandwidth).

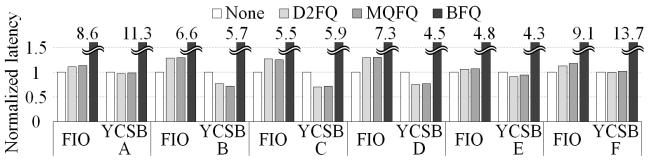


Figure 19: Normalized latency of the realistic workload (YCSB with FIO).

achieves high I/O bandwidth and low I/O latency whereas YCSB shows low I/O bandwidth and high I/O latency. On the contrary, D2FQ and MQFQ show smaller FIO bandwidths and larger YCSB bandwidths than None. Accordingly, both show longer I/O latency with FIO and shorter I/O latency with YCSB. BFQ shows very long I/O latency due to its inefficiency in I/O scheduling.

5 Related Work

Fair-share I/O Schedulers. Fair resource sharing is one of important goals of I/O resource sharing. Linux employs time slice-based fair schedulers, such as CFQ [3], BFQ [34], Argon [35] and FIOS [30]. Time slice-based schedulers are non-work conserving: I/O resources can remain unused while requests are available. Ahn et al. [2] proposed a budget-based fair share I/O scheduler implemented in Linux cgroup layer; it is also a non-work conserving scheduler. H-BFQ [28] has expanded the original BFQ to hierarchical cgroup structure. Fair queueing I/O schedulers [8, 9, 29] including D2FQ are work-conserving so they always try to keep I/O resource busy whenever requests exist. Fair queueing schedulers provide fairness using virtual time [8, 9, 29], and they controls the order of I/O requests to minimize the virtual time gap between any flows. With the advance in storage performance, it is necessary to dispatch multiple requests to a device to maximize the I/O performance. This relaxes the requirement of minimizing the virtual time gap between any flows. SFQ(D) [13] allows at most D outstanding requests. MQFQ [11] relaxes the requirement further to enable parallel dispatch with a little communication across cores. D2FQ also relaxes the assumption which is determined by the two thresholds (τ_m and τ_l) for a different reason: too small thresholds increases the tail latency by unnecessarily throttling requests.

Other I/O Schedulers. Lee et al. [20] isolate queues to prioritize reads over writes. Kyber [18] prioritize synchronous I/Os over asynchronous ones to foreground performance. Kim et al. [16] prioritize requests from foreground context holistically throughout the I/O stack. These schedulers, however, do not provide fair I/O resource management.

Scheduling Offloading to Device. FLIN [33] implements fair-share scheduler in the SSD firmware and identifies and considers the major sources of performance interference in a flash-based SSD. Joshi et al. [14] enlightens the use of NVMe WRR in Linux for SSD resource fairness. The use of NVMe WRR to the block cgroup is later implemented in the mainline Linux [27]. None of these work consider the sharing of queues with multiple flows, which is necessary when the number of flows exceeds the number of queues.

6 Conclusion

This paper proposes D2FQ, a low overhead high-performance fair-queueing I/O scheduler. D2FQ is carefully designed to implement the sophisticated fair-queueing I/O scheduler on top of the simple device-side I/O scheduling feature (i.e., NVMe WRR). Therefore, the CPU overhead associated with scheduling decision is minimized, thereby saving CPU cycles and improving I/O performance. Modern high-performance SSDs are changing the paradigm that the bottleneck is no longer I/O but CPU. We expect our light-weight fair-queueing scheme will help reduce the contention on the CPU and allow applications to use more CPU cycles for a useful way. We plan to extend our scheme to leverage the urgent priority class of the NVMe WRR for better quality of service of block I/O scheduling.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Janki Bhimani, for their valuable comments. This work was supported partly by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2020R1A2C2102406), by the MSIT (Ministry of Science and ICT), Korea, under the ICT Creative Consilience program (IITP-2020-0-01821) and by Samsung Electronics.

Availability

The source code is available at https://github.com/skkucsl/d2fq

References

[1] Samsung 980 Pro. https://news.samsung.com/global/samsung-delivers-next-level-ssd-performance-with-980-pro-for-gaming-and-high-end-pc-applications.

[2] Sungyong Ahn, Kwanghyun La, and Jihong Kim. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-Core Systems. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, page 111–115, USA, 2016. USENIX Association.

[3] Jens Axboe. Linux block IO—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[4] Budget Fair Queueing. https://lwn.net/Articles/784267/, 2019.

[5] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 33–48, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Data center bridging task group. http://www.ieee802.org/1/pages/dcbridges.html.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, August 1989.

[9] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. *SIGCOMM Comput. Commun. Rev.*, 26(4):157–168, August 1996.

[10] Jun He, Kan Wu, Sudarsun Kannan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Read as Needed: Building WiSER, a Flash-Optimized Search Engine. In *18th USENIX Conference on File and Storage Technologies*, FAST'20, pages 59–73, Santa Clara, CA, February 2020. USENIX Association.

[11] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-Queue Fair Queueing. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 301–314, USA, 2019. USENIX Association.

[12] Sitaram Iyer and Peter Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 117–130, New York, NY, USA, 2001. Association for Computing Machinery.

[13] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed Proportional Sharing for a Storage Service Utility. *SIGMETRICS Perform. Eval. Rev.*, 32(1):37–48, June 2004.

[14] Kanchan Joshi, Praval Choudhary, and Kaushal Yadav. Enabling NVMe WRR Support in Linux Block Layer. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'17, page 22, USA, 2017. USENIX Association.

[15] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. HIOS: A Host Interface I/O Scheduler for Solid State Disks. *SIGARCH Comput. Archit. News*, 42(3):289–300, June 2014.

[16] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 345–358, USA, 2017. USENIX Association.

[17] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the Performance of Fast NVM Storage with Udepot. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 1–15, USA, 2019. USENIX Association.

[18] Kyber multi-queue i/o scheduler. https://lwn.net/Articles/720071/, 2017.

[19] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-Latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 603–616, USA, 2019. USENIX Association.

[20] Minkyeong Lee, Dong Hyun Kang, Minho Lee, and Young Ik Eom. Improving Read Performance by Isolating Multiple Queues in NVMe SSDs. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, IMCOM '17, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*,

SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Yujie Liu, Victor Luchangco, and Michael Spear. Mindicators: A Scalable Approach to Quiescence. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, page 206–215, USA, 2013. IEEE Computer Society.

[23] MELLANOX TECHNOLOGIES. ConnectX-4 VPI. https://www.mellanox.com/files/doc-2020/pb-connectx-4-vpi-card.pdf.

[24] Mindicator. https://github.com/mfs409/nonblocking/tree/master/tsx_acceleration/mindicator.

[25] Multi-queue deadline I/O Scheduler. https://lwn.net/Articles/767987/, 2016.

[26] NVM express specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4a-2020.03.09-Ratified.pdf.

[27] Add support Weighted Round Robin for blkcg and nvme. https://lwn.net/Articles/810726/.

[28] Kwonje Oh, Jonggyu Park, and Young Ik Eom. H-BFQ: Supporting Multi-Level Hierarchical Cgroup in BFQ Scheduler. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp 20)*, pages 366–369. IEEE, 2020.

[29] Abhay Kumar Parekh. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Massachusetts Institute of Technology, 1992.

[30] Stan Park and Kai Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 13, USA, 2012. USENIX Association.

[31] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 475–488, USA, 2014. USENIX Association.

[32] Brent Stephens, Aditya Akella, and Michael M. Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 33–46, USA, 2019. USENIX Association.

[33] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 397–410. IEEE Press, 2018.

[34] Paolo Valente and Arianna Avanzini. Evolution of the BFQ Storage-I/O scheduler. In *2015 Mobile Systems Technologies Workshop*, pages 15–20. IEEE, 2015.

[35] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, page 5, USA, 2007. USENIX Association.

[36] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.

[38] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: Punching through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 477–492, USA, 2018. USENIX Association.