I/O control may be slowing your storage down dramatically

August 22, 2018

Many services, from WEB hosting and video streaming, to cloud storage, need to move data to/from storage. And they require that each per-client I/O flow be guaranteed a non-null bandwidth and a bounded latency. An expensive solution to provide these guarantees is to overprovision storage resources, so as to keep each resource underutilized, and thus have plenty of bandwidth available for the few I/O flows dispatched to each medium. Or one can control I/O. Throttling is a classical solution to attain this goal: it guarantees the desired bandwidth and latency to each flow, even on fully utilized drives. So problem solved: by controlling I/O with throttling, we can meet bandwidth and latency requirements using many fewer, highly utilized resources. Unfortunately, no.

Throttling does guarantee control even on drives that happen to be highly utilized, but, as we are going to see, it has a hard time actually making drives highly utilized. Even with greedy I/O flows, it easily ends up utilizing down to 20% of the available speed of a flash-based drive.

Such a speed loss may be particularly problematic with lower-end storage. On the opposite end, it is somewhat also disappointing with high-end hardware, as the Linux block I/O stack itself has been redesigned from the ground up [1], exactly to fully utilize the high speed of modern, fast storage. Finally, this problem affects distributed environments too, simply because it affects every node. In addition, throttling fails to guarantee the expected bandwidths if I/O contains both reads and writes, or ON/OFF sources.

On the bright side, there seems to be now an effective alternative for controlling I/O: the proportional-share policy, backed by the *bfq* I/O scheduler. It enables the utilization of storage speed to be pushed up to about 100%, at least with part of the problematic workloads for throttling. An improved version of *bfq* seems to also have chances to achieve this optimal result with most workloads. Finally, *bfq* guarantees bandwidths with all workloads. The current limitation of *bfq* is that its execution overhead becomes significant at speeds above 400 KIOPS, on commodity CPUs.

To describe the situation in more detail, we start by completing the list of the main cases where individual I/O flows must be guaranteed a low latency, a minimum bandwidth, or just some fairness:

- as mentioned above, a server, or a node in a data center, serving I/O for multiple clients, and possibly executing other administrative tasks at the same time;
- a host with several virtual machines or containers doing I/O at the same time;
- a personal system serving extra I/O in the background while some foreground I/O needs to be completed.

The performance of I/O control varies greatly depending on whether I/O flows constantly have pending I/O (for long). When this does not happen, the situation is rather rosy, even if extra workloads do saturate the device. For this lucky case, we survey the situation briefly in the next section. On the opposite end, the above throughput problems arise when flows are so many, or so heavy, that each flow has pending I/O, and thus requires bandwidth, for long. These problems are the focus of this article.

1 Low latency for lightweight, short I/O: all seems well

Through the *bfq* I/O scheduler [2], Linux can now guarantee a very low latency to lightweight flows containing sporadic, short I/O. No throughput issue arises, and no configuration is required. This capability benefits important time-sensitive tasks, such as playing or streaming videos or audios, as well as executing commands or starting applications. Up-to-date results on latency are available for storage ranging from slow to fast rotational devices [3, 4], and from embedded flash [5, 6, 7] to SATA [3, 8] and NVMe [9] SSDs.

Although benchmarks are not available yet, these guarantees might be provided also by the newly proposed *IO latency* cgroups controller [10]: it allows, but also imposes, to set target latencies for I/O requests of each group of processes, and privileges the groups with the lowest target latency.

2 Let's get started

As above stated, problems arise with heavier I/O flows. We analyze these problems through a set of tests. In the next sections we first introduce our testbed, i.e.,

- the system on which we executed our benchmarks;
- the main actors in our workloads: target and interferers;
- the workloads;
- the I/O policies we tested;
- the configurations we used for I/O policies;
- the quantities we measured.

Then we report our results, which can be reproduced very easily[1].

In particular, for conciseness, we shrank the next sections as much as possible. You can find missing information and more detailed comments in this document [11].

3 System and storage medium

We ran our tests with an *ext4* filesystem mounted on a PLEXTOR PX-256M5S SSD, which features a peak rate of ∼160MB/s with random I/O, and of ∼500 MB/s with sequential I/O. We used *blk-mq*, in Linux 4.18 (Ubuntu 18.04, although the distribution should have no influence on these tests). The system was equipped with a 2.4GHz Intel Core i7-2760QM CPU and a 1.3 GHz DDR3 DRAM. In such a system, a single thread doing synchronous reads reaches a throughput of 23 MB/s.

4 Target and Interferers

Multiple, prolonged I/O flows can generate highly variable mixes of I/O (reads+writes, sequential+random, constant+ON/OFF, ...). And the properties of a mix of I/O heavily influence the performance of a storage device, as well as the ability of I/O policies to guarantee bandwidths. Thus, to carry out a thorough analysis, one needs to generate a wide range of I/O mixes. As a feasible solution to achieve this goal, we use synthetic I/O flows in our tests. We make a distinct process generate each such I/O flow.

To provide bandwidth guarantees to the I/O of these processes (through I/O policies), we encapsulate each process in a distinct group. Groups belong to two categories:

Target A single-process, I/O-bound group, whose I/O we focus on. In particular, we measure the I/O throughput enjoyed by this group, to get the minimum bandwidth guaranteed to the group: the throughput of the group is at least as high as the minimum bandwidth guaranteed to the group.

Interferer A single-process group, whose role is to generate additional I/O that interferes with the I/O of the target.

Workloads contain one target and multiple interferers. Note that, even if we focus on only one target, we do consider multiple-flow configurations (as in, e.g., servers with multiple active clients). We analyze what happens to one generic flow, for as many as possible distinct combinations of workloads, as a feasible way to assess what would happen to any flow in any generic set of flows.

The single process in each group either reads or writes one file—different from the file read or written by any other process—after invalidating the buffer cache for the file. We use the following nomenclature for brevity. We call just *reader/writer* either the target or an interferer, depending on whether the single process in the target or the

[1] First `git clone https://github.com/Algodev-github/S.git`, then `cd S/run_multiple_benchmarks && sudo ./run_main_benchmarks.sh bandwidth-latency "low-none max-none prop-bfq"`

interferer performs the reading/writing of one file. Writers do buffered writes. We define a reader/writer as

- *random* or *sequential*, if it (the reader/writer) reads/writes its file at random positions or sequentially;
- *constant* or *sporadic* if it is constantly I/O-bound or if it randomly oscillates between ON and OFF I/O phases.

Basing on the last definition, we define a workload as *static* or *dynamic* if, respectively, all or not all readers and writers are constant. Finally, we define an interferer as *active* or *inactive* if it does or never does I/O during the test. When we mention a reader/writer interferer, we assume implicitly that such an interferer is *active*.

5 Workloads

The properties of an I/O mix depend on the properties of the I/O flows, and of the I/O stack that manipulates the mix: number of flows, I/O direction (reads/writes), spatial locality (sequential/random I/O), temporal locality (active/inactive, static/dynamic flows), I/O depth, I/O policy, and so on. It is practically impossible to test all combinations of the values of these parameters. So, we tried to cover at least the combinations that supposedly influence mostly the performance of the storage device and of the I/O policies. To this purpose, we defined the groups of generic workloads reported in Table 1.

For each group, the table reports the mixes of interferers that define that group. To get each workload, we considered, for each mix of interferers in the group, two possibilities for the target: it could be either a random or a sequential sync reader.

6 I/O policies

Linux provides two I/O-control mechanisms for guaranteeing (a minimum) bandwidth, or at least fairness, to long-lived flows: the *throttling* and the *proportional-share* I/O policies [12].

With throttling, one can set a maximum-bandwidth limit for the I/O of each group [12]. We call this limit *max limit*, for brevity. A group gets throttled if it issues I/O at a higher rate than its max limit. Max limits can be used to provide the service guarantee we focus on in this article, i.e., to guarantee minimum bandwidths to I/O flows. This result can be obtained in an indirect way: as a function of the total throughput available, per-group max limits can be set in such a way that there remains enough bandwidth available for each group. To this purpose, max limits must be sized, more precisely, according to the lowest total throughput that may happen to be available, i.e., according to the lowest peak rate that the device may happen to reach with the expected workloads.

Unfortunately, max limits have two evident drawbacks in terms of throughput. First, if some groups do not use their allocated bandwidth, that bandwidth cannot be reclaimed by other active groups. Second, unless expected workloads are known very precisely,

Table 1: Groups of workloads and corresponding interferer mixes. The target is either a random or a sequential constant sync reader.

Group of workloads	Set of interferers
Static sequential	Four constant synchronous sequential readers or four constant asynchronous writers, plus other five inactive interferers.
Static with varying randomness	Four constant synchronous readers or four constant asynchronous writers, with different degrees of I/O randomness, plus other five inactive interferers. As for the degree of randomness of readers and writers, one reader/writer is sequential, while the other three are random, with block sizes equal, respectively, to 4k, 128k and 1024k.
Static random	Four constant synchronous random readers, all with block size equal to 4k, plus other five inactive interferers.
Dynamic sequential/random	Two sporadic synchronous readers, one being random and one sequential; plus two sporadic synchronous writers, one random and one sequential; plus other five inactive interferers.

limits must comply with the very worst-case speed of the device, namely the device peak rate with purely random I/O. Such limits will evidently leave a lot of throughput unused with workloads that, if not limited, would make the device reach higher throughputs than just the random-I/O peak rate. To sum up, maximizing throughput is basically not a goal of max limits. Our results show the obvious consequences of the above two drawbacks.

Because of these drawbacks, an opposite, still experimental, *low limit* has been added to the throttling policy [13]. If a group is assigned such a low limit, then the throttling policy automatically, and dynamically, limits the I/O of the other groups in such a way to guarantee to the group a minimum bandwidth equal to its assigned low limit. In particular, this new throttling mechanism addresses the above throughput-loss drawbacks as follows: it throttles no group as long as every group is getting at least its assigned minimum bandwidth. We test this mechanism too. We do not consider also the very interesting problem of guaranteeing minimum bandwidths and, at the same time, enforcing maximum bandwidths.

The other I/O policy available in Linux, proportional share [12], targets weighted fairness. Each group is assigned a weight, and should receive a fraction of the total throughput proportional to its weight. This service scheme guarantees minimum bandwidths too, as low limits do in throttling. In particular, it guarantees to each group a minimum bandwidth equal to the ratio between the weight of the group, and the sum of the weights of all the groups that may be active at the same time.

The actual implementation of the proportional-share policy, on a given drive, depends on what flavor of the block layer is in use for that drive. If the flavor is legacy *blk*, then the policy is implemented by the *cfq* I/O scheduler. Unfortunately, *cfq* fails to control bandwidths with flash-based storage, especially on drive featuring command queueing. We do not consider this case at all in our tests. With *blk-mq*, proportional share is implemented by *bfq*. This is the combination we consider in our tests.

7 I/O-policy configurations

To benchmark both throttling (max and low limits) and proportional share, we tested, for each workload, the three pairs of I/O policies [12] and I/O schedulers reported in Table 2, with the configurations reported in the same table. In the end, we have three test cases for each workload. In addition, for some workloads, we considered two versions of *bfq* for the proportional-share policy.

One note before commenting on configurations: it is unfair, in many respects, to compare max limits with low limits and proportional share. Differently from the latter two solutions, max limits lack any mechanism for redistributing unused bandwidth. Yet the goal of this article is not to compare solutions basing on their similarity, but to survey all available solutions. From this point of view, max limits are probably the most popular solution for controlling bandwidth in Linux.

For throttling policies, we report results with only *none* as I/O scheduler, because results are the same with *kyber* and *mq-deadline*.

Table 2: Short names for pairs policy-scheduler, I/O policies, associated I/O schedulers, parameter configurations. *Parameter* stands for low limit, max limit or weight, depending on the policy. The same limits are set for both reads and writes.

Name	I/O policy	Scheduler	Parameter for target	Parameter for each of the four active interferers	Parameter for each of the five inactive interferers	Sum of parameters
low-none	Throttling with *low limits*	*none*	10MB/s	10 MB/s (tot: 40)	20 MB/s (tot: 100)	150 MB/s
max-none	Throttling with *max limits*	*none*	10MB/s	15 MB/s (tot: 60)	25 MB/s (tot: 125)	195 MB/s
prop-bfq	Proportional share	*bfq*	300	100 (tot: 400)	200 (tot: 1000)	1700

The capabilities of the storage medium drove policy configurations. *low-none* was the policy for which it was hardest to find a working configuration, i.e., a configuration for which the target low limit was actually met with at least the simplest workloads. So we first looked for a working configuration for *low-none*, and then configured the other policies accordingly.

If interferers can only be readers, then 10 MB/s is apparently about the maximum bandwidth that, on this drive, *low-none* can successfully guarantee to the target. In particular, *low-none* cannot guarantee more than 10MB/s if the target is a random reader, and interferers are sequential readers [14]. A configuration for which *low-none* provides this best-possible bandwidth guarantee is if we set to 10 MB/s the low limit for the target and for each active interferer. We used this configuration in our tests. Results remains the same regardless of the values used for *target latency* and *idle time*; at any rate, in these tests we set these parameters, respectively, to 100us and 1000us for every group. Finally, we set the low limits of the inactive interferers to twice the limits for active interferers, to possibly pose greater difficulties to the policy.

To get a fair comparison, we configured the other policies with the primary goal of guaranteeing the same minimum bandwidth as *low-none* to the target, in the same worst case as above. In particular, we chose the configurations that achieved this goal and, still for the sake of fairness, had the same ratio as *low-none* between the limits/weights of active and inactive interferers. In addition, to help *max-none* get the maximum possible throughput, we scaled interferer max limits so as to saturate the peak rate reached by the medium in the worst case, i.e., with random I/O (160 MB/s). More precisely,

considering that groups are unlikely to be all active at the same time in realistic scenarios, we overcommitted bandwidths a little bit, to make *max-none* get higher throughputs.

8 Quantities measured and statistics

We ran each workload ten times for each policy, plus ten times without any I/O control, i.e., with *none* as I/O scheduler and no I/O policy is use. For each run, we measured the I/O throughput of the target (which reveals the bandwidth guaranteed to the target), the cumulative I/O throughput of the interferers, and the total I/O throughput. These quantities fluctuated very little during each run, as well as across different runs. Thus in the graphs we report only averages over per-run average throughputs. In particular, for the case of no I/O control, we report only the total I/O throughput, to give an idea of the throughput that can be reached without imposing any control.

9 Results

A note before showing results: none of the throughput losses reported in this section have anything to do with I/O-request manipulation, such as request merging. These losses are due only to intrinsic issues of the I/O policies, which we highlight as we comment on results.

Figure 1 shows throughput results for the simplest group of workloads: *static sequential* (Table 1). With a random reader as target, against sequential readers as interferers, *low-none* does guarantee the configured low limit to the target. Yet it reaches only a very low total throughput, for the following reason. The throughput of the random reader evidently oscillates around 10MB/s during the test. This implies that it is at least slightly below 10MB/s for a significant percentage of the time. But when this happens, the low-limit mechanism limits the maximum bandwith of every active group to the low limit set for the group, i.e., to just 10 MB/s, as discussed in detail in [14].

The end result is a total throughput lower than 10% of the throughput reached without I/O control. Yet, the latter, very high throughput is reached by chocking the random I/O of the target, and serving almost only the throughput-boosting sequential I/O of the interferers. Then, it is probably more interesting to compare *low-none* throughput with the throughput reachable while actually guaranteeing 10MB/s to the target. The target is a single sync random reader, which reaches 23 MB/s while in service. So, to guarantee 10 MB/s to the target, it is enough to serve it for about half of the time, and the interferers for the other half. Since the device reaches ∼500MB/s with the sequential I/O of the interferers, the resulting throughput with this service scheme would be (500+23)/2 ∼ 260 MB/s. *low-none* then reaches less than 20% of the total throughput that could be reached while still preserving the target bandwidth.

As for *max-none*, the very low total throughput that *max-none* reaches with this first workload (one random reader against sequential readers) is due to the rigidity of the scheme: the bandwidth allocated for, but not used by the inactive interferers is not reclaimable. Even worse, the total throughput would become arbitrarily lower than

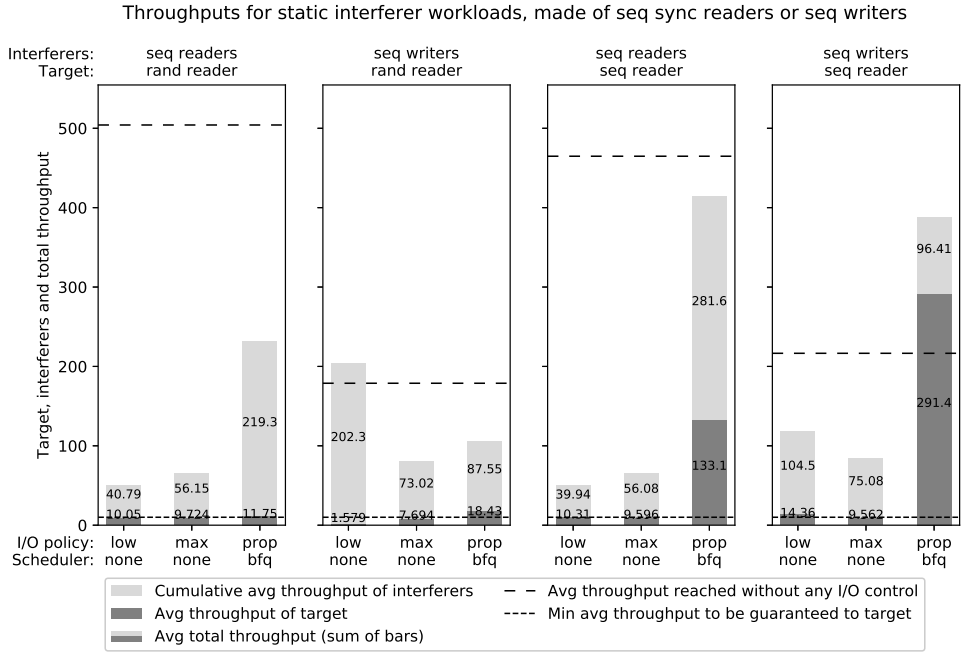

Figure 1: Results for the *static sequential* workloads.

that, if more interferers became inactive and/or maximum bandwidths were configured in unluckier ways.

Finally, *prop-bfq* provides the target with a slightly higher throughput than the other policies. This makes it harder for *prop-bfq* to reach a high total throughput, because *prop-bfq* serves more random I/O (from the target) than *low-none* and *max-none*. Nevertheless, *prop-bfq* gets a much higher total throughput than the other policies. According to the above estimate, this throughput is about 90% of the maximum throughput that could be reached, for this workload, without violating service guarantees. The reason for this good result is that *bfq* provides an effective implementation of the proportional-share service policy. At any time, each active group is granted a fraction of the current total throughput, and the sum of these fractions is equal to 1; so group bandwidths naturally saturate the available total throughput at all times.

Things change with the second workload: a random reader against sequential writers. Now *low-none* reaches a much higher total throughput than *prop-bfq*. Yet the reason is that *low-none* serves much more sequential (write) I/O than *prop-bfq*, because writes somehow break the low-limit mechanisms and prevail over the reads of the target. Conceivably, this happens because writes tend to both starve reads in the OS–mainly by eating all available I/O tags–and to cheat on their completion time in the drive. In contrast, *bfq* is intentionally configured to privilege reads, to counter these issues.

In particular, *low-none* gets an even higher throughput than without I/O control, because it serves the random I/O of the target even less than how *none* does without any throttling. A loss of guarantees happens at a lower extent with *max-none* too, but

without much gain in terms of total throughput (because of the max limits).

Finally, with both the last two workloads, *prop-bfq* reaches even higher total throughputs than with the first two workloads. It happens because also the target now does sequential I/O, and serving sequential I/O is much more beneficial for throughput than serving random I/O. With these two workloads, the total throughput is, respectively, close to or much higher than that reached without I/O control. For the last workload, the total throughput is much higher because, differently from *none*, *prop-bfq* privileges reads over asynchronous writes, and reads yield a higher throughput than writes. In contrast, *low-none* and *max-none* still get lower or even much lower throughputs than *prop-bfq*, because of the same issues that hindered throughput for these two policies with the first two workloads.

As for bandwidth guarantees, with readers as interferers (third workload), *prop-bfq* expectedly grants to the target a fraction of the total throughput proportional to its weight. In fact *bfq* approximates perfect proportional-share bandwidth distribution among groups doing I/O of the same type (reads or writes) and with the same locality (sequential or random). With the last workload, *prop-bfq* gives much more throughput to the reader than to all the interferers, because interferers are asynchronous writers, and *bfq* privileges reads.

Figure 2 reports results for flows with different degrees of I/O randomness: *static with varying randomness* (Table 1). These results basically match those for the *static sequential* workloads in Figure 1, except for an important point. With respect to the *static sequential* workloads, the total throughputs with *prop-bfq* decrease more or much more than with the other policies.

This decrease highlights the Achilles' heel of the *bfq* I/O scheduler. If the process in service does sync I/O and has a higher weight than some other process, then, to give strong bandwidth guarantees to the process, *bfq* plugs I/O dispatching every time the process remains temporarily without pending I/O requests. In this respect, processes actually have differentiated weights and do sync I/O in the workloads in Figure 2. So *bfq* systematically performs I/O plugging for them. Unfortunately, this plugging empties the internal queue(s) of the drive, which kills throughput with random I/O. And the I/O of some processes in these workloads is also random.

We defined the third group of workloads, *static random*, exactly to better highlight this important weakness of *bfq*. Figure 3 shows results for these workloads. The figure reports results not only for *bfq*, but also for an improved version of *bfq*, containing small changes to counter this weakness. This new version is currently under public testing, in the development branch of *bfq* [15] (where *bfq* for *blk-mq* is named *bfq-mq*).

As can bee seen, with only random readers, *prop-bfq* reaches an extremely lower total throughput than throttling policies. Still, the situation reverses with a sequential reader as target.

Yet, the most interesting results come from the improved version of *bfq*. It recovers most of the throughput loss with the workload containing only random I/O. Not only, with the workload with a sequential reader as target, it reaches about 3.7 times the total throughput of *max-none* or *low-none*.

We finish with the results for the last group of workloads, *dynamic sequential/random*,

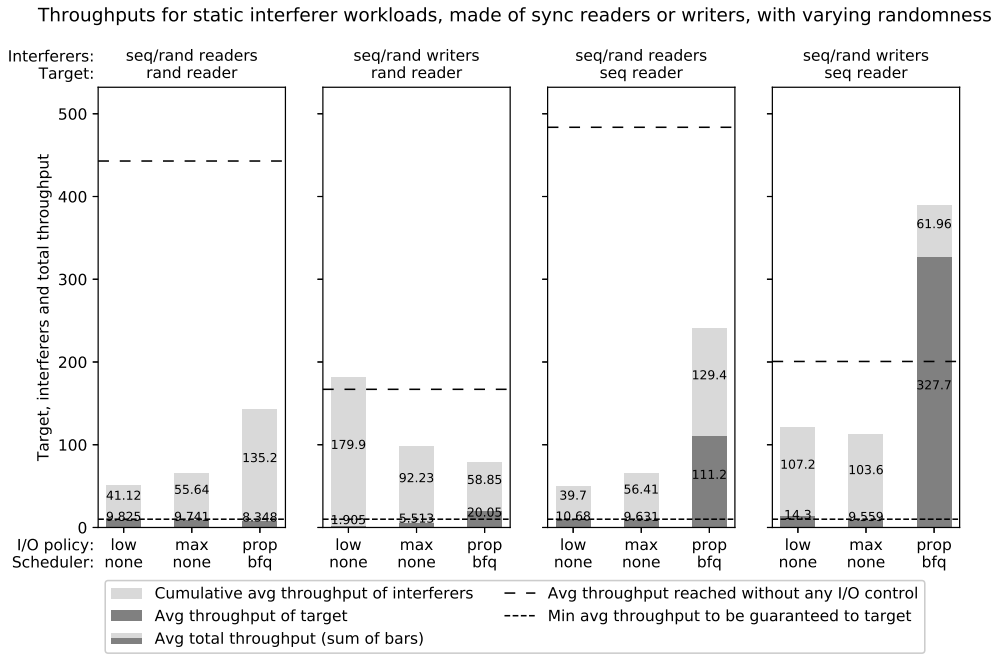

Figure 2: Results for the *static with varying randomness* workloads.

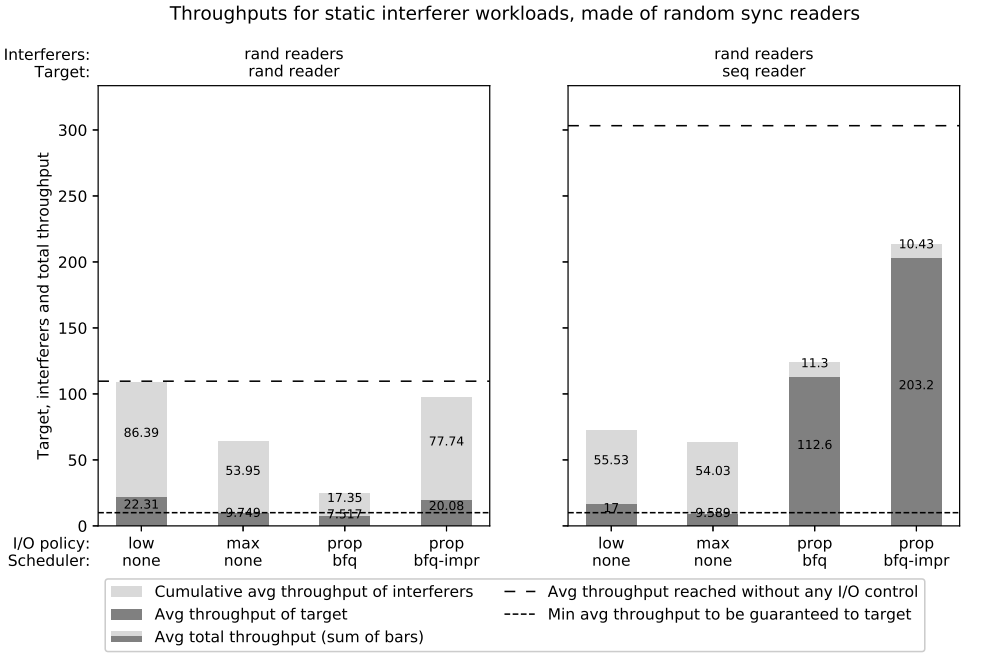

Figure 3: Results for the *static random* workloads.

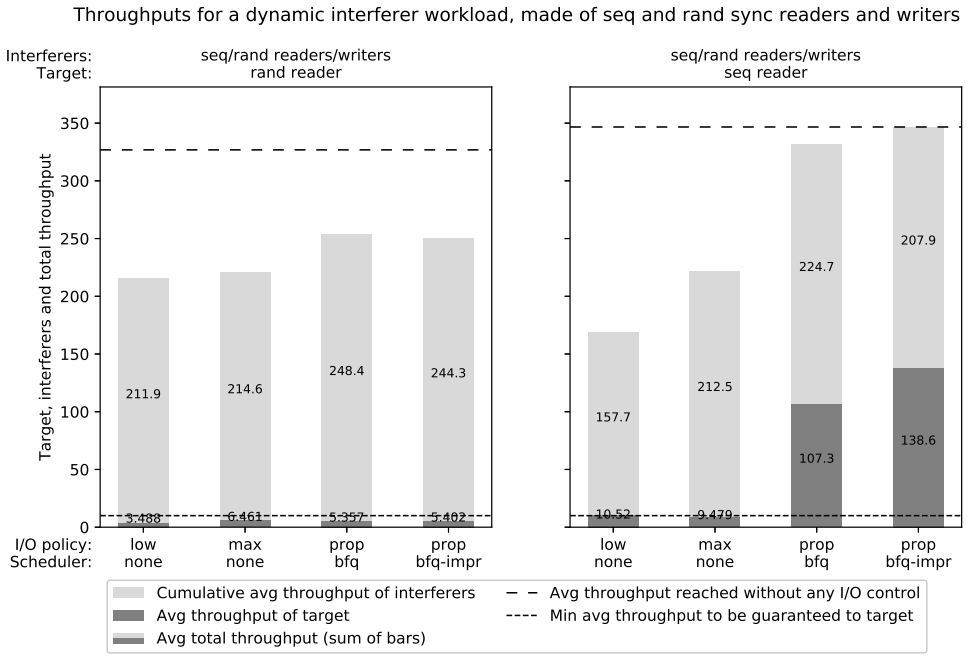

Figure 4: Results for the *dynamic sequential/random* workloads.

shown in Figure 4 (again also for the improved version of *bfq*). The goal of this group is to try to subsume the complexity of general, heterogenous and, above all, dynamic workloads. Results match those for static workloads, and highlight the same issues, apart from that no policy meets the minimum-bandwidth requirement for the target when the latter is a random reader. For *prop-bfq*, it is just a matter of assigning proper weights for this different workload.

10 Conclusion

When the main concern is the latency of flows containing short I/O, Linux seems now rather high performing, thanks to the *bfq* I/O scheduler and the *IO latency* controller. Yet, if the requirement is to guarantee explicit bandwidths, or just fairness, to I/O flows, then one must be ready to give up even most of the speed of the storage media. *bfq* helps with some workloads, but loses most of the throughput with workloads consisting of mostly random I/O. Fortunately, there is apparently hope for much better performance, as an improvement, still under development, seems to enable *bfq* to reach a high throughput with all workloads tested so far.

References

[1] [Online]. Available: https://lwn.net/Articles/552904

[2] [Online]. Available: https://www.kernel.org/doc/Documentation/block/bfq-iosched.txt

[3] [Online]. Available: http://algogroup.unimore.it/people/paolo/disk_sched/results.php

[4] [Online]. Available: https://youtu.be/ZeNbS0rzpoY

[5] [Online]. Available: https://elciotna18.sched.com/event/DXnF/a-solution-to-high-latencies-caused-by-io-paolo-valente-linaro

[6] [Online]. Available: https://youtu.be/gyM_JJtIvP0

[7] [Online]. Available: https://youtu.be/ANfqNiJVoVE

[8] [Online]. Available: https://youtu.be/1cjZeaCXIyM

[9] [Online]. Available: https://www.phoronix.com/scan.php?page=article&item=linux417-nvme-io&num=1

[10] [Online]. Available: https://lwn.net/Articles/758963/

[11] [Online]. Available: http://algogroup.unimore.it/people/paolo/pub-docs/extended-lat-bw-throughput.pdf

[12] [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt

[13] [Online]. Available: https://lkml.org/lkml/2017/1/14/310

[14] [Online]. Available: https://lkml.org/lkml/2018/4/24/365

[15] [Online]. Available: https://github.com/Algodev-github/bfq-mq