Evolution of the BFQ Storage-I/O Scheduler

Paolo Valente
Dipartimento di Fisica, Informatica e Matematica
Università di Modena e Reggio Emilia, Italy
paolo.valente@unimore.it

Arianna Avanzini
Dipartimento di Fisica, Informatica e Matematica
Università di Modena e Reggio Emilia, Italy
arianna.avanzini@unimore.it

Abstract—**An accurate storage-I/O scheduler, named Budget Fair Queueing (BFQ), was integrated with a special set of heuristics a few years ago. The resulting, improved scheduler, codenamed BFQ-v1, was able to guarantee a number of desirable service properties, including a high responsiveness, to applications and system services. In the intervening years, BFQ-v1 has become relatively popular on desktop and handheld systems, and has further evolved. But no official, comprehensive and concentrated documentation has been provided about the improvements that have followed each other. In this paper we fill this documentation gap, by describing the current, last version of BFQ (v7r8). We also show the performance of BFQ-v7r8 through some experimental results, in terms of throughput and application responsiveness, and on both an HDD and an SSD.**

Keywords. Storage scheduling, latency, interactive applications, soft real-time applications, throughput, fairness.

I. INTRODUCTION

An I/O scheduler for a shared storage device is a component in charge of deciding the order in which I/O requests (read/write) are dispatched to the device. The scheduler decides that order so as to achieve several goals[1], the main ones being typically:

- Achieving a high I/O throughput.
- Guaranteeing a *low latency* to time-sensitive tasks (see below).
- Guaranteeing the desired fraction of the I/O throughput to each application competing for the device.

As for the second main goal, two important classes of tasks are real-time and interactive ones.

Real-time tasks, i.e., tasks/applications that typically issue I/O requests spaced by at least a certain minimum inter-arrival period, and for which each I/O request should be completed within a given deadline. A common example is a multimedia player, for which a good scheduler is expected, e.g., to guarantee that frames are read in time to be played back without glitches.

Interactive tasks, i.e., tasks initiated by users, and such that users typically have to wait for the completion of these tasks before being able to perform next actions. Examples are: booting a system, starting an application, opening a file from within an application, and so on. Guaranteeing a low latency to interactive tasks means guaranteeing a high *responsiveness* to applications, or even to the overall system.

With respect to the above goals, BFQ is a proportional-share I/O scheduler [1] that allows each application to be guaranteed

the desired fraction of the I/O throughput, even if the latter fluctuates. This fraction is established by assigning a fixed weight to each application. BFQ also allows the device to achieve a high aggregate throughput.

BFQ-v1 is an enhanced version of BFQ [2], containing several improvements to better boost throughput and, above all, containing a few special heuristics to guarantee a high application and system responsiveness. The effectiveness of these heuristics is probably one of the reasons for the high popularity gained by BFQ. In this respect, BFQ has been adopted in several top Linux and Android distributions, plus several Linux-kernel variants for desktop and handheld systems. In addition, source BFQ patches are maintained as an independent, public free project [6].

Over the last years, BFQ has been not only maintained, but also constantly improved, till the current v7r8 version has been reached. A slightly previous version of BFQ has also been proposed to the Linux kernel development community [3]. The outcome of this proposal has been that BFQ is currently on the right track to replace CFQ, the default storage-I/O scheduler in most Linux distributions. In spite of this success, there is no official, single document describing all the important changes occurred in BFQ from its first v1 version to its last version.

A. Contribution

In this paper we describe the main BFQ changes from v1 to v7r8. These changes consists in:

- Improvements to the low-latency heuristics, aimed at achieving a higher and more stable responsiveness.
- A new *ad hoc* low-latency heuristic for soft real-time applications. One of the main benefits of this new heuristic is that, now, differently from BFQ-v1, a high responsiveness does not come at the price of an occasionally increased latency for soft real-time applications. In fact, soft real-time applications are now privileged too.
- A special strategy for handling NCQ-capable devices with respect to service guarantees: this strategy allows fairness and low-latency guarantees to be preserved also on these devices (this critical issue was only highlighted in [2], but not yet solved in BFQ-v1).
- Proper handling of flash-based devices and NCQ-capable HDD devices so as to maximize their throughput in all scenarios where this does not hurt fairness and low-latency guarantees.
- Proper handling of interleaved I/O workloads, which allows for higher throughput with QEMU virtual machines.

---

[1]Almost always in collaboration with other components of the layer of the operating system that handles storage [5].

Finally, to show the performance of BFQ-v7r8, we also report a selection of our experimental results, on both an HDD and an SSD, and concerning both throughput and responsiveness. In these experiments, BFQ is compared with all other standard Linux schedulers, that is: CFQ, DEADLINE and NOOP.

B. Organization of this paper

Section II introduces both the system model and the common definitions used in the rest of this paper. The original version of BFQ is briefly described in Section III, while Section IV outlines the additional heuristics and improvements (explained in depth in [2]) that turn the original version of BFQ into BFQ-v1. Section V is, instead, the core of the paper, describing the changes that led to BFQ-v7r8. Finally, Section VI reports experimental results.

II. SYSTEM MODEL AND COMMON DEFINITIONS

We consider a *storage system* made of a storage device, a set of *N* applications to serve and the BFQ scheduler in-between. The storage device is modeled as a sequence of contiguous, fixed-size *sectors*, each identified by its *position* in the sequence.

The storage device serves two types of *I/O requests*: reading and writing a set of contiguous sectors. We say that a request is *sequential/random* with respect to another request if the first sector (to read or write) of the request is/is not located just after the last sector of the other request.

At the opposite end, requests are issued by the *N* applications, which represent the possible entities that can compete for access to the storage device in a real system, as, e.g., *threads* or *processes*. We define the set of pending requests for an application as the *backlog* of the application. We say that an application is *backlogged* if its backlog is not empty, and *idle* otherwise. For brevity, we denote an application as *random/sequential* if most times the next request it issues is random/sequential with respect to the previous one. We say that a request is *synchronous* if the application that issued it can issue its next request only after this request has been completed. Otherwise we denote the request as *asynchronous*. We say that an application is *receiving service* from the storage system if one of its requests is currently being served.

III. THE BFQ ALGORITHM

In this section we describe a simplified version of the original BFQ algorithm. In particular, after describing the logical scheme of BFQ, we provide a few details about how BFQ boosts the throughput with sequential synchronous I/O requests (see [1], [2] for more details). The logical scheme of BFQ is depicted in Figure 1. Solid arrows represent the paths followed by the requests until they reach the storage device. There is an internal request queue for each application, where the latter inserts its requests by invoking the interface *add_request()* function. We define the set of requests present in one of these queues as the *backlog* of the application owning the queue. We say that an application is *backlogged* if its backlog is not empty and *idle* otherwise. Access to the storage device is granted to one application at a time, denoted as the *in-service application*. Each application has a *budget* assigned
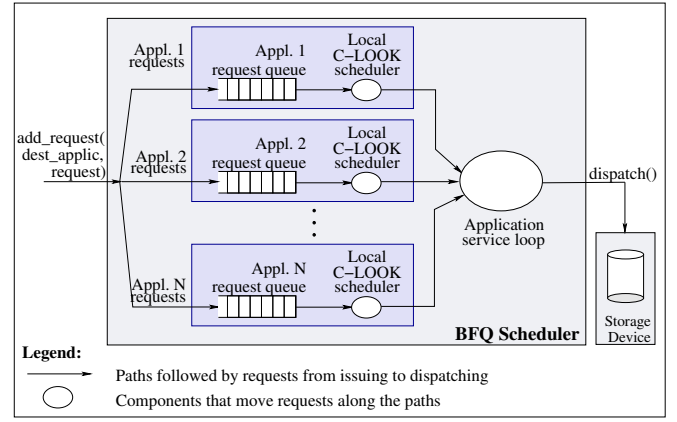

Figure 1: Logical scheme of BFQ.

to it, measured in number of sectors. When an application becomes the in-service one, it is served exclusively until either this budget it exhausted or the backlog of the application empties (the application becomes idle). Then BFQ selects the new in-service application, and so on. In other words, the in-service application cannot be preempted until one of the above two events occurs. In more detail, each time BFQ turns from not having any application backlogged to having at least one application backlogged, the *application service loop* depicted in Figure 1 starts. This loop, repeated until there is at least one backlogged application, can be sketched as follows:

1) **Choice of the next in-service application.** The internal fair-queueing scheduler, called B-WF^{2}Q+ [1], [2], chooses the next in-service application among the backlogged ones. See below for more details on B-WF^{2}Q+.

2) **Request dispatch.** The loop blocks, waiting for the storage device to invoke the *dispatch()* function. When there is at least one backlogged queue, operating system mechanisms around BFQ guarantee that the device driver will shortly invoke this function. When this happens:

   a) The local C-LOOK scheduler chooses the request to serve among those waiting in the queue of the in-service application; this request is extracted from that queue and dispatched to the storage device (right side of Figure 1). Note that C-LOOK is effective with both rotational and non-rotational devices, as both achieve maximum throughput with sequential I/O.

   b) The budget of the application is decremented by the size of the dispatched request.

   c) If the budget of the application is exhausted or the application has no more backlog, jump to step 3, otherwise repeat step 2.

3) **Application deactivation and budget recomputation.** The application stops being the in-service one, and is assigned a new budget. A simple feedback-loop algorithm is used to compute the new budget.

Concerning the first step of the *application service loop*, it is worth stressing that B-WF^{2}Q+ guarantees to each application a fraction of the throughput *independent of the size of the budgets* assigned to the application. This property may seem counterintuitive at a first glance, because the larger the budget

assigned to an application is, the longer the application will use the device once granted access to it. But B-WF^{2}Q+ basically balances this fact by postponing the service of an application in proportion to the budget currently assigned to the application. In this respect, being free to choose budgets without affecting throughput shares is one of the key properties that enables BFQ to provide, at the same time, both accurate guarantees and a high throughput (see [1], [2] for further details).

Boosting the throughput with sequential synchronous requests

According the step 2.c of the *application service loop*, when an application becomes idle, BFQ deactivates it and starts serving a new application. However, if the last request of the application was synchronous, then the application may be *deceptively* idle, as it may be already preparing the next request and may issue it shortly. In fact, a minimum amount of time is needed for an application to handle a just-completed synchronous request and to submit the next one.

For this reason, when an application becomes idle but its last request was synchronous, BFQ actually does not deactivate the application and hence does not switch to another application. In contrast, in this case BFQ *idles the device* and waits, for a time interval in the order of the seek and rotational latencies, for the possible arrival of a new request from the same application. The purpose of this wait is to allow a possible next sequential synchronous request to be waited for and sent to the device as it arrives. Though apparently counterintuitive, on rotational devices this wait usually results in a boost of the device throughput [4] with sequential and synchronous applications. In this respect, note that most mainstream applications issue synchronous requests.

As shown in [1], device idling is instrumental also in preserving service guarantees with synchronous requests. On flash-based devices, the throughput with random I/O is high enough to make idling detrimental at a first glance. But most operating systems perform *readahead*, which makes idling effective also on these devices.

IV. BFQ-v1

This section outlines the improvements, introduced in the original version of BFQ, that led to BFQ-v1. Full details can be found in [2]. This outline will help us motivate and better describe the changes that led finally to BFQ-v7r8 (Section V).

A. Low latency for interactive applications

As highlighted in the introduction, a system is responsive if it starts applications quickly and performs the tasks requested by interactive applications just as quickly[2]. This fact motivates the first step of the event-driven heuristic presented in this subsection and called just *low-latency* heuristic hereafter.

1) To privilege the I/O of a just-created application, the weight of its associated queue is raised by multiplying

---

[2]All other system-related factors being equal, the minimum time needed to start a given application, or to complete a given task, depends on the type and amount of I/O involved, and on the speed of the storage device. In particular, such a minimum time interval is extremely variable: as can be seen, e.g., from our experimental results in [8], it may range from tens of milliseconds to several seconds. See Section V-A for how BFQ deals with this issue.

it by a *weight-raising coefficient C_{rais}*. The initial I/O of the application is most likely due to the application loading itself, thus guaranteeing a higher fraction of the throughput to this I/O will speed up exactly the loading of the application.

2) The weight of the queue is linearly decreased while the application receives service, until it becomes again equal to initial value.

3) While *weight-raised*, the queue unconditionally enjoys device idling every time it empties; in fact, if the requests of a queue are synchronous, then performing device idling for the queue is a necessary condition to guarantee that the queue receives a fraction of the throughput proportional to its weight (see [2] for details).

In addition, consider that any interactive application blocks and waits for user input both after starting up and after executing some task. After a while, the user may trigger new operations, after which the application stops again, and so on. Accordingly, the low-latency heuristic *weight-raises* again a queue in case it becomes backlogged after being idle for a sufficiently long time. The weight-raising then lasts for the same time as for a just-created application.

B. Other improvements

This subsection briefly lists other improvements, introduced in BFQ-v1, which are basically unchanged in BFQ-v7r8, and which do not need to be described in detail to understand the features of BFQ-v7r8. Full details on these improvements can be found in [2].

A new peak rate estimator was introduced to smooth out spikes which caused estimation errors on the device peak rate (which in its turn is used in computing the maximum possible budget assigned to applications).

Budget-assignment rules were re-tuned so as to converge to large budgets more quickly and to decrease the worst-case latency experienced by first requests of applications.

More fairness to random and slow applications was provided by making less stringent the rules used to limit device utilization for these types of applications.

A write overcharge coefficient was introduced to avoid starvation of read requests in the presence of burts of write requests (problem caused by the internal caching of write requests performed by devices).

V. BFQ-v7r8

This section describes in some detail the main improvements and extra heuristics implemented on top of BFQ-v1, and leading to BFQ-v7r8. We have already provided a summary of these changes in Section I-A.

A. New low-latency heuristic

The low-latency heuristic of BFQ-v7r8 is identical to that of BFQ-v1, apart from the following differences. In BFQ-v1, the low-latency heuristic performs a smooth decrease of the weight of a weight-raised queue, until the weight reaches its original value. Instead, in BFQ-v7r8 the heuristic is more aggressive: it lets a weight-raised queue keep its weight constantly equal

to *orig_weight* * C_{rais}, for the full duration of its *weight-raising period* (period computed as detailed below). In addition, the device-idling timeout for a weight-raised queue is raised as well: this reduces the probability that an application is deactivated because it performs synchronous requests and its next request does not arrive in time. Reducing this probability is fundamental to make more sure that, also in the presence of deceptive idleness, the application receives the desired, high fraction of the throughput.

Finally, the weight-raising period for an application is automatically computed

- according to the device speed and type (rotational or non-rotational), and
- so as to be equal to the time needed to load (start up) a large-size application on that device, with cold caches and with no additional background workload.

B. Low latency for soft real-time applications

To guarantee a low latency also to the I/O requests issued by soft real-time applications, BFQ-v7r8 sports an additional, dedicated heuristic, which weight-raises also the queues associated to applications deemed as soft real-time. To be deemed as soft real-time, an application must meet two requirements. First, the application must not require an average bandwidth higher than the approximate bandwidth required to playback or record a compressed high-definition video. Second, the request pattern of the application must be *isochronous*, i.e., after issuing a request or a batch of requests, the application must stop issuing new requests until a certain, minimum amount of time, say Δ, has elapsed from when all the pending requests of the application have been completed. After that, the application may issue a new batch, and so on.

Actually, the second requirement is stronger that the standard isochrony requirement, because, according to the latter, an application is still deemed as isochronous even if it issues new requests *immediately* after its last pending request is completed. BFQ-v7r8 uses, instead, the above stronger requirement to prevent also greedy (i.e., I/O-bound) applications from being incorrectly deemed, occasionally, as soft real-time. In fact, if *any amount of time*, including zero, is fine, then even a greedy application may, paradoxically, meet both the above bandwidth and isochrony requirements, if: (1) the application performs random I/O and/or the device is slow, and (2) the CPU load is high. The reason is the following. First, if condition (1) is true, then, during the service of the application, the throughput may be low enough to let the application meet the bandwidth requirement. Second, if condition (2) is true as well, then the application may occasionally behave in an apparently isochronous way, because it may simply stop issuing requests while the CPUs are busy serving other processes.

With proper values for Δ, the strong isochrony requirement of BFQ-v7r8, on one side, filters out the above possible false positives, because greedy applications issue *all* of their requests as quickly as they can. On the other side, it lets actual soft real-time in, because the latter always spend some time processing data after each batch of requests is completed.

In particular, the heuristic works as follows. First, in view of the isochrony requirement, the heuristic checks whether an application may be soft real-time (and therefore gives to the application the opportunity to be deemed as such) *only* when both the following two conditions happen to hold: 1) the queue associated with the application has expired and is empty, 2) there is no outstanding request of the application. Then, suppose that both conditions hold at time, say, *t_c* and that the application issues instead its next request at time, say, t_i. At time t_c the heuristic computes the next time instant, *soft_rt_next_start*, such that, only if $t_i \geq$ *soft_rt_next_start*, then both the above bandwidth and isochrony requirements are met, i.e., such that, in particular, $t_i \geq t_c + \Delta$ holds.

The current value of Δ is a little bit higher than the value, 8 ms, that we have found, experimentally, to be adequate on a real, general-purpose machine. We had to add 4 system *ticks*[3] to this base value, to make the filter quite precise also in slower, embedded systems, and in KVM/QEMU virtual machines.

If the application actually issues its next request after time *soft_rt_next_start*, then its associated queue will be weight-raised for a relatively short time interval. If, during this time interval, the application proves again to meet the bandwidth and isochrony requirements, then the end of the weight-raising period for the queue is moved forward, and so on. Note that an application whose associated queue never happens to be empty when it expires will never have the opportunity to be deemed as soft real-time.

C. Preserving an accurate service and a high throughput on modern drives

BFQ-v7r8 implements some special strategies to preserve, also on modern drives, two of its main service properties: accurate service guarantees (including low latencies) and a high throughput. As for the first service property, modern storage devices usually implement Native Command Queueing (NCQ), which boosts the throughput by prefetching and internally reordering requests. I/O schedulers typically allow NCQ-capable drives to prefetch I/O requests to achieve maximum throughput. Unfortunately, as discussed in detail and shown experimentally in [2], this may cause fairness and latency guarantees to be violated. The main problem is that the internal scheduler of an NCQ-capable drive may postpone the service of some unlucky (prefetched) requests as long as it deems serving other requests more appropriate to boost the throughput.

In order to address this issue, which was still open as of [2], BFQ-v7r8 always grants device idling to weight-raised queues, even if the device supports NCQ. This allows BFQ to start serving a new queue, and therefore allows the drive to prefetch new requests, only after the idling timeout expires. At that time, all the outstanding requests of the expired queue have been most certainly served.

On the opposite end, idling may be detrimental for aggregate throughput, because it prevents the drive from pre-fetching many requests and hence achieving maximum performance. For this reason BFQ-v7r8 performs idling only when strictly needed. In particular, BFQ-v7r8 disables idling for all symmetric workloads from the point of view of weights (i.e., in which each

---

[3]A *tick* is the finest time resolution in a Linux system, in that it is equal to the inverse of the frequency of the system timer. A *tick* typically ranges from 1 to 10 ms.

application has to receive the same fraction of the throughput). Indeed, with these workloads each application tends to receive about the same fraction of the throughput, as actually desired, even if idling is disabled. In addition, possible deviations from a perfectly fair service are typically not critical in symmetric scenarios. In fact, if doing I/O, time-sensitive applications would have been weight-raised by BFQ, thereby making the scenario most certainly asymmetric.

Rotational devices need extra care with respect to flash-based storage devices: if applications perform sequential I/O, then idling is vital to achieve maximum throughput, even with NCQ. Therefore, with rotational devices, BFQ-v7r8 may disable idling only for applications performing random I/O, even if the device is NCQ-capable.

D. Early Queue Merge

A set of applications may happen to perform interleaved reads, i.e., requests whose union would give rise to a sequential read pattern. This happens, e.g., with QEMU/KVM, which splits the I/O generated by a guest into multiple chunks, and lets these chunks be served by a pool of processes, iteratively assigning the next chunk of I/O to the first available process.

To achieve a high throughput also with interleaved reads, BFQ-v7r8 adopts a mechanism named Early Queue Merge (EQM), whose purpose is to get a sequential read pattern out of interleaved read requests. EQM checks every newly arrived request against the next request of the in-service application, and, if the two requests are close, interprets this fact as an indication that the application that issued the newly-arrived request and the in-service application are likely to be performing interleaved I/O. In this case, EQM merges the queues associated to the two applications. Once the two queues are merged, the C-LOOK scheduler in Step 2.a of the application service loop in Section III guarantees that the service order of the requests in the queue resulting from the merge is as sequential as possible.

Finally, EQM is implemented in such a way to preserve the low-latency properties of BFQ, by properly keeping the weight-raising state of merged queues.

VI. EXPERIMENTAL RESULTS

In this section we report a selection of our last test results, with BFQ-v7r8, CFQ, DEADLINE and NOOP, under Linux v4.0.0, and on the following two devices:

- A (high-speed) Seagate ST1000DM003 HDD
- A PLEXTOR PX-256M5S SSD

For each device, we report throughput and application-responsiveness results. As for responsiveness, we report only our results on the cold-cache start-up time of a heavy commonly-used application, *oowriter* (Open Office Writer).

We present only this selection of results for space constraints. Full results for many more test cases (including responsiveness with other applications, latency for soft real-time applications, throughput with interleaved reads, and so on), and on other devices (including, e.g., RAIDs and eMMCs) can be found in [8]. In all these other combinations of test cases and devices, the relative performance among the schedulers is about the same as in the selection presented here. We have obtained all
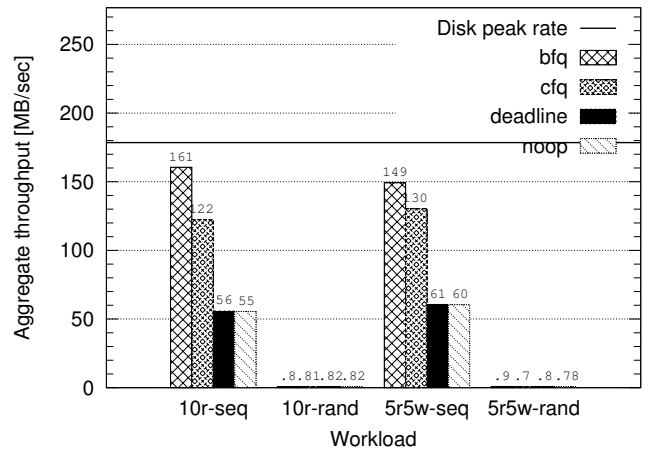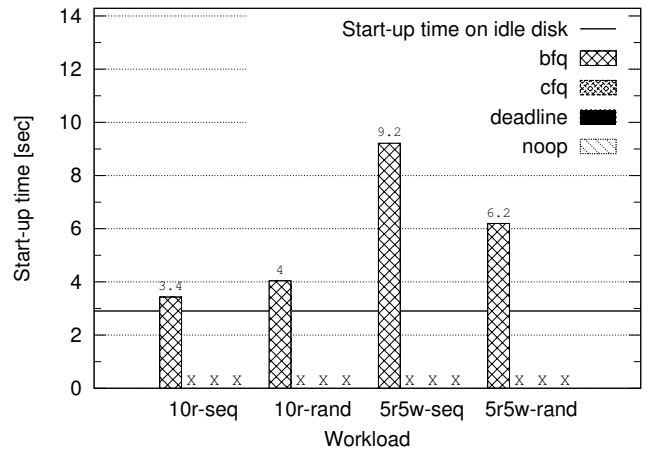

Figure 2: Throughput on the Seagate HDD.

Figure 3: *oowriter* start-up time on the Seagate HDD.

these results with the benchmark suite described in [2] and available at [7].

A. Seagate HDD

Figure 2 shows the throughput achieved by each scheduler while one of the following four heavy workloads is being executed: 10 parallel sequential or random readers (*10r-seq*, *10r-rand*), 5 parallel sequential or random readers plus 5 parallel sequential or random writers (*5r5w-seq*, *5r5w-rand*). BFQ outperforms the other schedulers with the sequential workloads, especially DEADLINE and NOOP. BFQ outperforms the other schedulers also with *5r5w-rand*, but in this case the maximum gap is with respect to CFQ. With *10r-rand* all the schedulers achieve, instead, about the same performance.

Figure 3 shows the cold-cache start-up time of *oowriter* while one of the above heavy background workloads is being executed. The symbol X means that, for that workload and with that scheduler, the application failed to start in 60 seconds. Except for testcases including writers, BFQ again guarantees a start-up time comparable to that achieved with an idle device,
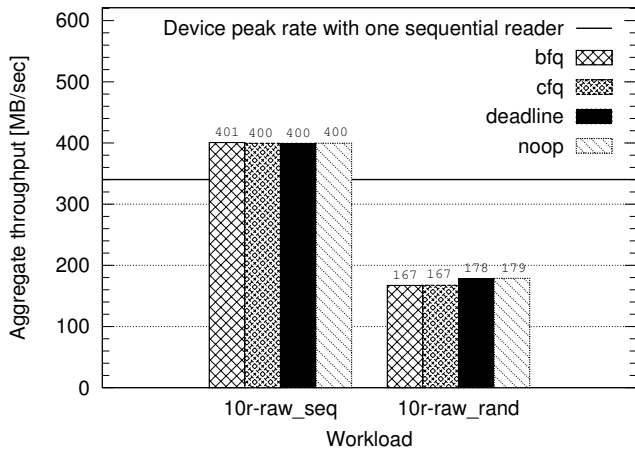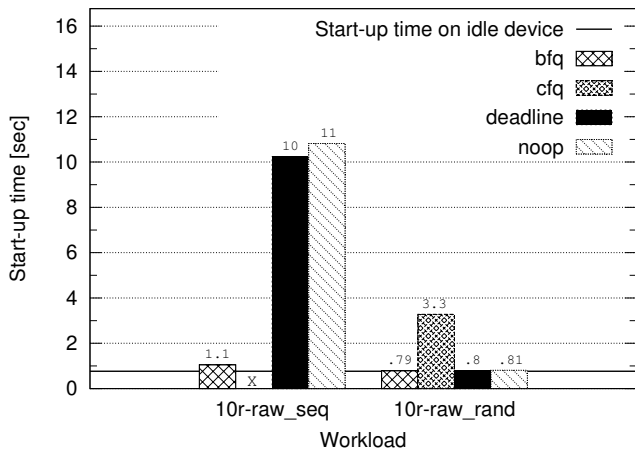
Figure 4: Throughput on the Plextor SSD.

Figure 5: *oowriter* start-up time on the Plextor SSD.

readers, the throughput achieved by BFQ is the same as the other schedulers. On the other hand, BFQ loses about 6 percent of throughput with random readers, for the following reason. With random readers, the number of IOPS is extremely higher, and all CPUs spend all their time either executing instructions or waiting for I/O (the total idle-time percentage is 0). Therefore, the processing time of I/O requests influences the maximum throughput achievable. As a conclusion, the throughput slightly grows as the complexity, and hence the execution time, of the schedulers decrease.

As for responsiveness, as shown in Figure 5, BFQ achieves almost the lowest-possible start-up time with both workloads. The high start-up times with the other schedulers in the presence of sequential readers is a consequence also of the fact that, to maximize throughput, the device prefetches requests, and, among internally-queued requests, privileges sequential ones. As explained in Section V-C, BFQ instead prevents the device from prefetching requests when that would hurt responsiveness, low latency or, in general, fairness guarantees.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have described BFQ-v7r8 and its improvements with respect to BFQ-v1. According to our experimental results and to the figures of merits we considered, in most cases BFQ-v7r8 outperforms existing, production-quality Linux schedulers. Still, BFQ has further important challenges to face. First, to cope with high-speed devices, a lighter version of BFQ may need to be defined. Secondly, a new, multi-queue software architecture has been devised in the Linux kernel to keep up with the speed of new-generation, fast storage devices (more precisely, this new architecture has been devised in the Linux *block layer*, where a scheduler like BFQ resides). To preserve service guarantees also in an architecture where I/O requests flow through multiple, independent queues, BFQ may need somehow to be turned into a distributed algorithm. Finally, to preserve low-latency guarantees also in the presence of bursts of asynchronous write requests, the well-known asynchronous-write issues of modern operating systems need to be somehow addressed. While these issues are not caused by the storage-I/O schedulers themselves, the latter could become part of possible solutions to the problem.

REFERENCES

[1] P. Valente and F. Checconi, *High throughput disk scheduling with fair bandwidth distribution.* IEEE Transactions on Computers 59.9 (2010). http://algogroup.unimore.it/people/paolo/disk_sched/bfq-techreport.pdf

[2] P. Valente and M. Andreolini, *Improving application responsiveness with the bfq disk I/O scheduler.* Proceedings of the 5th Annual International Systems and Storage Conference. ACM, 2012. http://algogroup.unimore.it/people/paolo/disk_sched/bfq-v1-suite-results.pdf

[3] https://lkml.org/lkml/2014/5/27/314

[4] S. Iyer and P. Druschel, *Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O.* ACM SIGOPS Operating Systems Review. Vol. 35. No. 5. ACM, 2001.

[5] M. Bjørling et al. *Linux block IO: introducing multi-queue SSD access on multi-core systems.* Proceedings of the 6th International Systems and Storage Conference. ACM, 2013.

[6] http://algogroup.unimore.it/people/paolo/disk_sched/

[7] http://algogroup.unimore.it/people/paolo/disk_sched/benchmark-suite.php

[8] http://algogroup.unimore.it/people/paolo/disk_sched/results.php

whereas with all the other schedulers the application fails, with any background workload, to start in 60 seconds. As for testcases including writers, the higher start-up time of *oowriter* with BFQ is mainly due to well-known issues with asynchronous writes, issues little related with the storage-I/O scheduler at hand. In more detail, *oowriter* needs to perform some asynchronous writes (probably on a log file) to complete its start-up process. Because of the high rate at which the five greedy writers issue write requests, dirty pages stack up in the page cache: after that, and until greedy writers stop, also asynchronous writes become blocking, i.e., also a process issuing an asynchronous write request is blocked until that request is completed. This significantly inflates the start-up time of *oowriter*, independently of the storage-I/O scheduler.

B. PLEXTOR SSD

As can be seen in Figure 4, with the SSD we considered only raw readers, i.e., processes reading directly from the device. We made this choice to avoid writing large files repeatedly, and hence wearing out the device too quickly. With sequential