Improving Application Responsiveness with the BFQ Disk I/O Scheduler

Paolo Valente
Dipartimento di Ingegneria dell'Informazione
Università di Modena e Reggio Emilia
paolo.valente@unimore.it

Mauro Andreolini
Dipartimento di Ingegneria dell'Informazione
Università di Modena e Reggio Emilia
mauro.andreolini@unimore.it

Abstract

BFQ (Budget Fair Queueing) is a production-quality, proportional-share disk scheduler with a relatively large user base. Part of its success is due to a set of simple heuristics that we added to the original algorithm about one year ago. These heuristics are the main focus of this paper.

The first heuristic enriches BFQ with one of the most desirable properties for a desktop or handheld system: responsiveness. The remaining heuristics improve the robustness of BFQ across heterogeneous devices, and help BFQ to preserve a high throughput under demanding workloads. To measure the performance of these heuristics we have implemented a suite of micro and macro benchmarks mimicking several real-world tasks, and have run it on three different systems with a single rotational disk. We have also compared our results against Completely Fair Queueing (CFQ), the default Linux disk scheduler.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Secondary storage; D.4.8 [*Performance*]: Measurements.

General Terms Experimentation, measurement, performance.

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

1. Introduction

BFQ is a proportional-share disk scheduler [1] that allows each application to be guaranteed the desired fraction of the disk throughput, even if the overall throughput fluctuates. At the same time, BFQ achieves a high disk throughput and guarantees a low latency to applications performing little and sporadic I/O, such as multimedia applications. BFQ has been implemented by Checconi and Valente in the Linux kernel, and is publicly available [2].

Thanks to the above properties, BFQ allows a user to enjoy the smooth playback of a movie while downloading other files, or while some service is accessing the disk. Unfortunately, BFQ may not achieve the same good performance in terms of *responsiveness*, i.e., time to load, and hence to start, applications, and time to complete the batches of I/O requests that interactive applications issue sporadically. For example, what is the start-up time of a large application on a loaded disk? If the application is guaranteed the same fraction of the disk throughput as the other applications (as it usually happens), and if many other applications are competing for the disk, then transferring all the sectors needed to load the application may take a long time.

A further important limitation of the original version of BFQ is that it has been thoroughly tested on just one system, equipped with a low-end disk (to make sure that the disk was the only bottleneck), and under workloads generated by at most five processes reading one private file each. The robustness and the effectiveness of BFQ should be verified across heterogeneous systems, including also RAIDs and solid-state drives (SSD), and against both a higher number of concurrent requests and a sudden increase of the workload.

Finally, in previous work [1] an important problem has been highlighted theoretically: with most mainstream applications, disk-drive internal queueing—such as Native Command Queueing (NCQ)— should cause both fairness and latency guarantees to be violated with any scheduler, including BFQ. However, experimental evidence was still missing.

Contributions of this paper

In this paper we report our contributions for overcoming the above limitations of BFQ:

- A set of simple heuristics added to BFQ to improve responsiveness, preserve a high throughput under demanding workloads and improve the robustness with respect to heterogeneous systems. Hereafter we call BFQ+ the resulting new version of BFQ.

SYSTOR '12 June 4–6, Haifa, Israel.

- A suite of micro and macro benchmarks [2] mimicking several real-world tasks, from reading/writing files in parallel to starting applications on a loaded disk, to watching a movie while starting applications on a loaded disk.
- A detailed report of the experimental results collected by running the above suite with BFQ+, BFQ and CFQ [14], on three Linux systems with a single rotational disk. One system was NCQ-capable, and we ran our experiments also with a FIFO scheduler on it.

In our experiments we did not consider either schedulers aimed only at throughput boosting or real-time and proportional-share research schedulers. The reason is that these schedulers may suffer from more or less obvious latency problems, as discussed in §5. Addressing these issues is out of the scope of this paper.

Our results with BFQ+ can be summarized as follows: differently from CFQ and regardless of the disk load, interactive applications now experience almost the same latency as if the disk was idle. At the same time, BFQ+ achieves up to 30% higher throughput than CFQ under most workloads. The low latency of interactive applications is achieved by letting them receive more than their fair share of the disk throughput. Nevertheless, the heuristic fits the original accurate service provided by BFQ well enough to still guarantee that non-interactive, time-sensitive applications (e.g., video players) experience a worst-case latency not higher than 1.6 times that experienced under CFQ.

The scheduling decisions made by BFQ+ comply with keeping a high throughput also with flash-based devices (§3). This fact and, above all, the new low-latency features described in this paper have made BFQ+ appealing to smartphones as well. In general, BFQ+ has been adopted in a few Linux distributions and is currently the default disk scheduler in some Linux-kernel variants as well as in a variant of Android. See [2] for more information.

As for disk-drive internal queueing, our results show that NCQ does affect service guarantees as foreseen in [1], up to the point of making a system unusable. Finally, we are investigating the impact of RAIDs and SSDs, see the conclusions.

Organization of the paper

In §2 we introduce both the system model and the common definitions used in the rest of the paper. BFQ is then described in §3, while the proposed heuristics can be found in §4. Finally, after describing the related work and the problems caused by disk-drive internal queueing in §5, we describe the benchmark suite and report our results in §6.

2. System model and common definitions

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

The disk device serves two types of *disk 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. This definition of a random request is only formal: the further the first sector of the request is from the last sector of the reference request, the more the request is random in real terms.

Requests are issued by the N applications, which represent the possible entities that can compete for disk access 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 *sequential* or *random* if most times the next request it issues is sequential or random with respect to the previous one, respectively. 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.

3. The original BFQ algorithm

In this section we outline the BFQ algorithm. A more detailed description is available in this technical report [3], whereas a full description can be found in the original paper on BFQ [1]. BFQ+ is identical to BFQ, apart from that it also contains the heuristics described in §4.

BFQ grants exclusive access to the disk to each application for a while, and implements this service model by associating every application with a *budget*, measured in number of sectors. After an application is selected for service, its requests are dispatched to the disk one after the other, and the budget of the application is decremented by the size of each request dispatched. The application is *deactivated*, i.e., its service is suspended, only if one of the following three events occurs: 1) the application finishes its budget, 2) the application becomes idle and its last request was asynchronous, 3) a special *budget timeout* fires (§3.3).

When an application is deactivated, BFQ performs two actions. First, it assigns a new budget to the application. This budget is calculated using a simple feedback-loop algorithm, described in detail in §3.2. Second, BFQ chooses the next application to serve through an internal fair-queueing scheduler, called B-WF^{2}Q+ and described in some detail in §3.1.

When an application becomes idle but its last request was synchronous, BFQ does not deactivate the application. In contrast, it *idles the disk* and waits for the possible arrival of a new request from the same application. In particular, BFQ waits for a time interval in the order of the seek and rotational latencies. The purpose is to allow a possible next sequential synchronous request to be sent to the disk as it arrives. On rotational devices, this wait usually results in a

boost of the disk throughput [4]. Disk idling is instrumental also in preserving service guarantees with synchronous requests [1]. 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.

In contrast, idling the disk to wait for the arrival of a random request usually provides little or no benefits on both rotational and non-rotational devices. Hence BFQ automatically disables disk idling for random applications.

3.1 B-WF^{2}Q+ and service properties

Under BFQ each application is associated with a fixed *weight*, and B-WF^{2}Q+ schedules applications in such a way that each application receives, in the long term and *regardless of the budgets assigned to the application*, a fraction of the disk throughput equal to the weight of the application, divided by the sum of the weights of the other applications competing for the disk. If an application is not assigned a weight explicitly, then BFQ sets the weight of the application to a common, system-wide value.

The above guarantee on throughput distribution may seem counterintuitive at first glance, because the larger the budget assigned to an application is, the longer the application will use the disk once granted access to it. But B-WF^{2}Q+ basically balances this fact, because the larger the budget assigned to the application is, the longer B-WF^{2}Q+ postpones the service of the application (see the original paper on BFQ [1] for full details and proofs). Besides, as shown in §3.2, the budgets assigned to an application are *independent of the weight* of the application.

As for short term guarantees, B-WF^{2}Q+ guarantees that each request is completed with the minimum possible worst-case delay with respect to when the request would be completed in an ideal perfectly-fair system (more precisely BFQ guarantees the minimum possible worst-case delay for a budget-by-budget service scheme). In more detail, the worst-case delay guaranteed by BFQ to the requests of an application is given by the sum of two components, with each component proportional to, respectively: 1) the maximum budget that BFQ may assign to any application, and 2) the maximum possible difference between the budget that BFQ may assign to the application and the actual number of sectors that the application *consumes* before it becomes idle. It follows that, the *tighter* the budgets assigned to the application are, the lower the second component of the delay is. Assigning, in general, a small budget to any application would instead keep low both components. These issues are taken into account in the budget-assignment algorithm described in the next subsection.

3.2 Budget assignment

The decoupling between the budgets assigned to an application and the fraction of the throughput guaranteed to the application gives BFQ an important degree of freedom: for each application, BFQ can choose, without perturbing throughput reservations, the budget that presumably best boosts the throughput or fits the application's requirements.

First, to achieve a high throughput many sequential requests must be performed. In this respect, when a new application is selected for service, its first request is most certainly random with respect to the last request of the previous application under service. As a result, depending on the hardware, the access time for the first request of the new application may range from 0.1 ms to about 20 ms. In the best case, after this random access all the requests of the new application are sequential, and hence the disk works at its peak rate (more precisely, for a rotational disk, the peak rate for the zone interested by the I/O). However, since the throughput is zero during the access time for the first request, the average disk throughput gets close to the peak rate only after the application has been served continuously for a long enough time. In the end, to achieve a high disk throughput, sequential applications should be assigned large enough budgets. To put into context, even with a worst-case access time of 20 ms, the average throughput reaches $\sim$90% of the peak rate after 150 ms of continuous service.

In contrast, assigning small budgets to applications improves service guarantees, as it reduces the delay mentioned in §3.1. These facts are at the heart of the feedback-loop algorithm of BFQ for computing budgets: each time an application is deactivated, the next budget of the application is increased or decreased so as to try to converge to a value equal to the number of sectors that the application is likely to request the next time it becomes active. However, the assigned budgets can grow up to, at most, a disk-wide *maximum budget* B_{max}. BFQ computes/updates B_{max} dynamically. Especially, BFQ frequently samples the disk peak rate, and sets B_{max} to the number of sectors that could be read, at the estimated disk peak rate, during a disk-wide, user configurable, *maximum time slice* T_{max}. The default value of T_{max} is 125 ms, which, according to the above estimates, is enough to get maximum throughput even on average devices. We describe in more detail the feedback-loop algorithm and the disk peak rate estimator in §4.2 and §4.3, where we show how, by enhancing these components, we improve the service properties and increase the throughput under BFQ.

3.3 Preserving fairness with random requests

BFQ imposes a time constraint on disk usage: once an application has been granted access to the disk, the application is served for at most T_{max} time units (T_{max} is the maximum time slice defined in §3.2), after which a *budget timeout* fires, and the application is deactivated, even if it has still backlog.

This falling back to time fairness prevents random applications from holding the disk for a long time and substantially decreasing the throughput. To further limit the extent at which random applications may decrease the throughput, on a budget timeout BFQ also (over)charges the just deactivated application an entire budget even if the application

has used only part of it. This reduces the frequency at which applications incurring budget timeouts access the disk.

4. Proposed heuristics

In addition to low responsiveness on loaded disks, in our experiments with BFQ we have also found the following problems: slowness in increasing budgets if many disk-bound applications are started at the same time, incorrect estimation of the disk peak rate, excessive reduction of the disk utilization for applications that consume their budgets too slowly or that are random only for short time intervals, and tendency of disk writes to starve reads. The heuristics and the changes reported in the following subsections address these problems. Each heuristic is based on one or more static parameters, which we have tuned manually. According to our experiments, and to the feedback from BFQ+ users, it seems unlikely that an administrator would have to further tune these parameters to fit the system at hand.

4.1 Low latency for interactive applications

A system is responsive if it starts applications quickly and performs the tasks requested by interactive applications just as quickly. This fact motivates the first step of the event-driven heuristic presented in this subsection and called just *low-latency* heuristic hereafter: the weight of any newly-created application is raised to load the application quickly. The weight of the application is then linearly decreased while the application receives service.

If the application is interactive, then it will block soon and wait for user input. After a while, the user may then trigger new operations after which the application stops again, and so on. Accordingly, as shown below, the low-latency heuristic raises again the weight of an application in case the application issues new requests after being idle for a sufficiently long (configurable) time.

In the rest of this subsection we describe the low-latency heuristic in detail and discuss its main drawback: the low-latency heuristic achieves responsiveness at the expense of fairness and latency of non-interactive applications (as, e.g., soft real-time applications). Trading fairness or latency of soft real-time applications for responsiveness may be pointless in many systems, such as most servers. In this respect, under BFQ+ the low-latency heuristic can be dynamically enabled/disabled through a *low_latency* parameter.

When a new application is created, its original weight is immediately multiplied by a *weight-raising coefficient* C_{rais}. This lets the application get a higher fraction of the disk throughput, in a time period in which most of its requests concern the reading of the needed portions of executables and libraries. The initial raising of the weight is shown in the topmost graph in Fig. 1, assuming that the application is created at time t_0 and that its original weight is w. The graph also shows the subsequent variation of the weight, which is described below. The bottommost graph shows in-

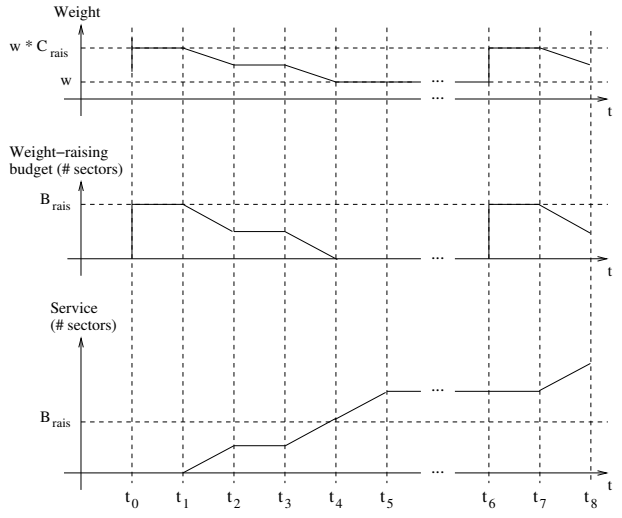


Figure 1: Weight raising for an application created at time t_0, become idle at time t_5 and again backlogged at time t_6; the application is served only during $[t_1, t_2]$, $[t_3, t_5]$ and $[t_7, t_8]$.

stead the amount of service received by the application (do not consider the graph in the middle for a moment).

If a new application keeps issuing requests after its start up is accomplished, then preserving a high weight would of course provide no further benefit in terms of start-up time. Unfortunately, a disk scheduler does not receive any notification about the completion of the loading of an application. To address this issue, BFQ+ decreases the weight of an application linearly while the application receives service, until the weight of the application becomes equal to its original value. This guarantees that the weight of a disk-bound application drops back smoothly to its original value.

To compute the slope at which its weight is decreased, an application is also associated with a *weight-raising budget*, set to an initial value B_{rais} when the application is created. As shown in the middle graph in Fig. 1, while an application enjoying weight raising is served (intervals $[t_1, t_2]$ and $[t_3, t_5]$), this special budget is decremented by the amount of service received by the application, until it reaches 0 (time t_4). Also the weight of the application is linearly decreased as a function of the service received, but with such a slope that it becomes again equal to its original value exactly when the weight-raising budget is exhausted (time t_4). In formulas, for each sector served, the weight is decremented by $\frac{C_{rais}-1}{B_{rais}}w$, where w was the original value of the weight.

After the weight-raising budget is exhausted, the weight of the application remains unchanged ($[t_4, t_5]$). But, if the application becomes backlogged after being idle for a configurable *minimum idle period* T_{idle} ($[t_5, t_6]$), then the weight of the application is again multiplied by C_{rais} and the application is assigned again a weight-raising budget equal to B_{rais} (time t_6). The weight and the weight-raising budget of the application are then again decremented while

the application receives service ($[t_6, t_8]$), as in the case of a newly-created application.

As already noted, disk idling is instrumental in preserving service guarantees in the presence of synchronous requests [1]. Accordingly, to make sure that the applications whose weight is being raised do enjoy a low latency even if they perform random I/O, BFQ+ does not disable disk idling for these applications, whatever their request patterns are. There is however a time constraint, whose purpose is, in contrast, to prevent random applications from keeping a high weight and hence harming the disk throughput for too long. An application must consume its weight-raising budget within a configurable *maximum raising time* T_{rais} from when its weight is raised. If this time elapses, the weight-raising budget is set at zero and the weight of the application is reset to its original value.

After some tuning, we set the above parameters to the minimum values sufficient to achieve a very low start-up time even for as large applications as the ones in the OpenOffice suite: $C_{rais} = 10$, $B_{rais} = 24$ MB, $T_{rais} = 6$ sec, $T_{idle} = 2$ sec. We are also investigating ways for adjusting all or part of these parameters automatically.

Raising the weight of interactive applications is a straightforward solution to reduce their latency with any weight-based scheduler. The crucial point is what the consequences on non-interactive long-lived applications are. In fact, non-interactive long-lived applications do not benefit from any weight raising and are therefore penalized if other applications can get more throughput than their fair share.

The user of a desktop may be willing to tolerate a temporary drop of throughput for long-lived best-effort applications, as file download or sharing, in return of a definitely higher system responsiveness. In contrast, few users would be happy if their long-lived soft real-time applications, as, e.g., audio players, suffered from perceptible quality degradation. Hence only a small increase in latency can be accepted for these applications. Fortunately, the service properties of BFQ+ come into play exactly in this respect: the effectiveness of BFQ+ in reducing the latencies of soft real-time applications balances the tendency of the heuristic to increase the same latencies. We investigated this important point in our experiments, and discovered that, with the above values of C_{rais} and B_{rais}, non-interactive time-sensitive applications—as, e.g., video players—are still guaranteed latencies comparable to the ones they enjoy under CFQ (also thanks to the smaller initial budget now assigned to applications, §4.3).

4.2 A new peak rate estimator

As showed in §3.2, the maximum budget B_{max} that BFQ/BFQ+ can assign to an application is equal to the number of sectors that can be read, at the estimated peak rate, during T_{max}. In formulas, if we denote as R_{est} the estimated peak rate, then $B_{max} = T_{max} * R_{est}$. Hence, the higher R_{est} is with respect to the actual disk peak rate, the higher is the probability that applications incur budget timeouts unjustly (§3.3). Besides, a too high value of B_{max} degrades service properties unnecessarily (§3.1).

The peak rate estimator is executed each time the application under service is deactivated after being served for at least 20 ms. The reason for not executing the estimator after shorter time periods is filtering out short-term spikes that may perturb the measure. The first step performed by the estimator in BFQ is computing the disk rate during the service of the just deactivated application. This quantity, which we can denote as R_{meas}, is computed by dividing the number of sectors transferred, by the time for which the application has been active. After that, R_{meas} is compared with R_{est}. If $R_{est} < R_{meas}$, then $R_{est} \leftarrow R_{meas}$.

Unfortunately, our experiments with heterogeneous disks showed that this estimator is not robust. First, because of Zone Bit Recording (ZBR), sectors are read at higher rates in the outer zones of a rotational disk. For example, depending on the zone, the peak rate of the MAXTOR STM332061 in Table 1 ranges from 55 to about 90 MB/s. Since the estimator stores in R_{est} the maximum rate observed, ZBR may easily let the estimator converge to a value that is appropriate only for a small part of the disk. Second, R_{est} may *jump* (and remain equal) even to a much higher value than the maximum disk peak rate, because of an important, and difficult to predict, source of spikes: hits in the disk-drive cache, which may let sectors be transferred in practice at bus rate.

To smooth the spikes caused by the disk-drive cache and try to converge to the actual average peak rate over the disk surface, we have changed the estimator as follows. First, now R_{est} may be updated also if the just-deactivated application, despite not being detected as random, has not been able to consume all of its budget within the *maximum time slice* T_{max}. This fact is an indication that B_{max} is too large. Since $B_{max} = T_{max} * R_{est}$, R_{est} is probably too large as well and should be reduced.

Second, to filter the spikes in R_{meas}, a discrete low-pass filter is now used to update R_{est} instead of just keeping the highest rate sampled. The rationale is that the average peak rate of a disk is a relatively stable quantity, hence a low-pass filter should converge more or less quickly to the right value. The new estimator is then:

```
if (applic_service_time >= 20 ms)
  if (R_est < R_meas or
      (not applic_is_random and not budget_exhausted))
    R_est = (7/8) * R_est + (1/8) * R_meas;
```

The $7/8$ value for α, obtained after some tuning, did allow the estimator to effectively smooth oscillations and converge to the actual peak rate with all the disks in our experiments.

4.3 Adjusting budgets for high throughput

As already said, BFQ uses a feedback-loop algorithm to compute application budgets. This algorithm is basically a set of three rules, one for each of the possible reasons why

an application is deactivated. In our experiments on aggregate throughput, these rules turned out to be quite slow to converge to large budgets with demanding workloads, as, e.g., if many applications switch to a sequential, disk-bound request pattern after being non-disk-bound for a while. On the opposite side, BFQ assigns the maximum possible budget B_{max} to a just-created application. This allows a high throughput to be achieved immediately if the application is sequential and disk-bound. But it also increases the worst-case latency experienced by the first requests issued by the application (§3.2), which is detrimental for an interactive or soft-real time application.

To tackle these throughput and latency problems, on one hand we changed the initial budget value to $B_{max}/2$. On the other hand, we re-tuned the rules, adopting a multiplicative increase/linear decrease scheme. This scheme trades latency for throughput more than before, and tends to assign high budgets quickly to an application that is or becomes disk-bound. The description of both the new and the original rules follows.

No more backlog. In this case, the budget was larger than the number of sectors requested by the application, hence to reduce latency the old rule was simply to set the next budget to the number of sectors actually consumed by the application. In this respect, suppose that some of the requests issued by the application are still outstanding, i.e., dispatched to the disk device but not yet completed. If (part of) these requests are also synchronous, then the application may have not yet issued its next request just because it is still waiting for their completion. The new rule considers also this sub-case, where the actual needs of the application are still unknown. In particular: if there are still outstanding requests, the new rule does not provide for the budget to be decreased, on the contrary the budget is doubled *proactively*, in the hope that: 1) a larger value will fit the actual needs of the application, and 2) the application is sequential and a higher throughput will be achieved. If instead there is no outstanding request, the budget is decreased linearly, by a small fraction of the maximum budget B_{max} (currently $1/8$). This is the only case where the budget is decreased.

Budget timeout. In this case, increasing the budget would provide the following benefits: 1) it would give the chance to boost the throughput if the application is basically sequential, even if the application has not succeeded in using the disk at full speed (because, e.g., it has performed I/O on a zone of the disk slower than the estimated average peak rate), 2) if this is a random application, increasing its budget would help serving it less frequently, as random applications are also (over)charged the full budget on a budget timeout. The original rule did set the budget to the maximum value B_{max}, to let all applications experiencing budget timeouts receive the same share of the disk time. In our experiments we verified that this sudden jump to B_{max} did not provide sensible practical benefits, rather it increased the latency of applications performing sporadic and short I/O. The new, better performing rule is to only double the budget.

Budget exhaustion. The application has still backlog, as otherwise it would have fallen into the no-more-backlog case. Moreover, the application did not cause either a disk-idling timeout or a budget timeout. As a conclusion, it is sequential and disk-bound: the best candidate to boost the disk throughput if assigned a large budget. The original rule incremented the budget by a fixed quantity, whereas the new rule is more aggressive, and multiplies the budget by four.

4.4 More fairness towards temporarily random and slightly slow applications

We found, experimentally, the following three situations to occur frequently. First, if too little time has elapsed since a sequential application has started doing I/O, then the positive effect on the throughput of its sequential accesses may not have yet prevailed on the throughput loss occurred while moving the disk head onto the first sector requested by the application. Second, some applications generate really few, yet very far, random requests at the beginning of a new disk-bound phase, after which they start doing sequential I/O. Third, due to ZBR, an application may be deemed slow when it is performing I/O on the slowest zones of the disk. BFQ considers these applications harmful for the throughput, and hence adopts the following heuristic towards them: to reduce the disk utilization of these applications, BFQ (over)charges them with a full budget, and/or disables disk idling for them. We found that this heuristic of BFQ causes throughput loss and worse responsiveness. None of these countermeasures is taken by BFQ+ in any of the above three situations.

4.5 Write throttling

One of the sources of high I/O latencies and low throughput under Linux, and probably under any operating system, is the tendency of write requests to starve read ones. The reason is the following. Disk devices usually signal the completion of write requests just after receiving them. In fact, they store these requests in the internal cache, and then silently flush them to the actual medium. This usually causes possible subsequent read requests to starve. The problem is further exacerbated by the fact that, on several file systems, some read operations may trigger write requests as well (e.g., access-time updating).

To keep low the ratio between the number of write requests and the number of read requests served, we just added a *write (over)charge coefficient*: for each sector written, the budget of the active application is decremented by this coefficient instead of one. As shown by our experimental results, a coefficient equal to ten proved effective in guaranteeing high throughput and low latency.

5. Related work

We can broadly group the schedulers aimed at providing a predictable disk service as follows: 1) real-time sched-

ulers [5–8]; 2) proportional-share timestamp-based schedulers [9–12]; and 3) proportional-share round-robin schedulers [13, 14]. Unfortunately, real-time and timestamp-based proportional-share research schedulers may suffer from low throughput and degradation of the service guarantees with mainstream applications, as shown in detail in [1]. In brief, the service guarantees of these schedulers hold if the arrival time of every request of any application is independent of the completion time of the previous request issued by the same application. The problem is that this property just does not hold for most applications on a real system.

As for round-robin schedulers, we can use the production-quality CFQ disk scheduler as a reference to describe the main properties of the schedulers in this class. Differently from BFQ+, CFQ grants disk access to each application for a fixed time slice (as BFQ basically does only for random applications, §3.3). Slices are scheduled in a round-robin fashion and disk idling is performed as in BFQ+. Unfortunately, disk-time fairness may suffer from unfairness in throughput distribution. Suppose that two applications both issue, e.g., sequential requests, but for different zones of the disk. Due to ZBR, during the same time slice an application may have a higher/lower number of sectors served than the other. Another important fact is that under a round-robin scheduler any application may experience, independently from its weight, $O(N)$ worst-case delay in request completion time with respect to an ideal perfectly fair system. This delay is much higher than the one of BFQ+ (§3.1). Finally, also CFQ exports a *low_latency* configuration parameter: when enabled, CFQ tries to reduce the latency of interactive applications in a similar vein as BFQ+.

It is worth mentioning also schedulers aimed only at achieving a high disk throughput [4, 15]. They provide basically no latency guarantee, as they may not serve an application for a long time if, e.g., a sequential access is being performed in parallel (they may however perform well under symmetric workloads [16]). For space limitations we cannot describe also scheduling frameworks, we discuss them in this technical report [3]. The issues related to internal queueing are instead described in the next subsection.

5.1 Disk-drive internal queueing

If multiple disk-bound applications are competing for the disk, but are issuing only synchronous requests, and if the operating-system disk scheduler performs disk idling for synchronous requests, then a new request is dispatched to the disk only after the previous one has been completed. As a result, a disk-drive internal scheduler cannot do its job (fetch multiple requests and reorder them). Both CFQ and BFQ+ address this issue by disabling disk idling altogether when internal queueing is enabled.

As also shown by our experimental results, NCQ provides little or no advantage with all but purely random workloads, for which it actually achieves a definite throughput boost. On the other hand, our results show that the price paid for this benefit is loss of throughput distribution and latency guarantees, with any of the schedulers considered, at such an extent to make the system unusable. The causes of this problem are two-fold. The first is just that, once prefetched a request, an internal scheduler may postpone the service of the request as long as it deems serving other requests more appropriate to boost the throughput. The second, more subtle cause has been pointed out in [1], and regards a high-weight application issuing synchronous requests. Such an application, say A, may have at most one pending request at a time, as it can issue the next request only after the previous one has been completed (we rule out *readahead*, which does not change the essence of this problem). Hence the backlog of A empties each time this request is dispatched. If the disk is not idled and other applications are backlogged, any scheduler would of course serve another application. As a consequence, the application A would not obtain the high share of the disk throughput (or disk time) it should receive.

6. Benchmark suite and experimental results

In this section we show the results of our performance comparison among BFQ, BFQ+, CFQ and FIFO on rotational disks, with and without NCQ. We consider FIFO only in our experiments with NCQ, because FIFO is commonly considered the best option to get a high throughput with NCQ. Under Linux the FIFO discipline is implemented by the NOOP scheduler. In the next subsection we show the software and hardware configurations on which the experiments have been run, and discuss the choice of the subset of our results that we report in this paper. The experiments themselves are then reported in the following subsections. These experiments concern aggregate throughput, responsiveness and latency for soft real-time applications. Especially, the last index is measured through a video-playing benchmark. For each experiment we highlight which heuristics contributed to the good performance of BFQ+. To this purpose, we use the following abbreviations for each of the heuristics reported in §4.1–4.5: *H-low-latency*, *H-peak-rate*, *H-throughput*, *H-fairness* and *H-write-throt*.

To perform the experiments reported in the following subsections we prepared a suite of benchmarks that mimic, or are actually made of, real-world I/O tasks. The suite is available here [2]. We describe each benchmark at the beginning of each subsection. Actually, the suite contains more benchmarks than those described in this paper. A complete description of the suite and of the motivations for which we have defined an *ad hoc* new suite instead of using an existing one can be found in [3]. That document also contains our results with code-development applications, not reported in this paper for space limitations. All the results and statistics omitted in this paper and in [3] can instead be found in [2]. In addition, the benchmark suite contains the general script that we used for executing the experiments reported in this paper (all these experiments can then be repeated easily).

6.1 Test bed and selected results

To verify the robustness of BFQ+ across different hardware and software configurations, we ran the benchmark suite under Linux kernel releases ranging from 2.6.32 to 3.1, and on the three systems with a single rotational disk shown in Table 1. On each system, the suite was run twice: once with a standard configuration, i.e., with all the default services and background processes running, with the purpose of getting results close to the actual user experience; and once with an *essential* configuration, i.e., after removing all background processes, with the goal of removing as much as possible any source of perturbations not related to the benchmarks.

For both schedulers, we used the default values of their configuration parameters. In particular, for BFQ+ and BFQ, the maximum time slice T_{max} was equal to 125 ms (§3.2). For CFQ, the time slice was equal to 100 ms and *low_latency* was enabled, whereas, for BFQ+ the benchmarks have been run with *low_latency* both enabled and disabled (§4.1). Unless otherwise stated, for BFQ+ we report our results with *low_latency* enabled, and highlight interesting differences with the other case only when relevant.

The relative performance of BFQ+ with respect to BFQ and CFQ was essentially the same under any of the kernels, on any of the systems and independently of whether a standard or *essential* configuration was used. Besides, for each system and kernel release we collected a relatively large number of statistics, hence, for brevity, for the experiments without NCQ and apart from the video-playing benchmark, we report our results only for the 2.6.34 kernel on the third system. We chose this system because its disk speed and software configuration are closer to an average desktop system with respect to the other two. As for video playing, we report our results on the first system instead. Since this system is the one with the slowest disk, it allows us to show more accurately the performance degradation of BFQ+ with respect to BFQ and CFQ.

Regarding NCQ, as shown in Table 1 we had only one system with this feature, and we report here our results under the 2.6.34 kernel on that system. As previously stated, with NCQ we ran the benchmarks also with NOOP. In the next subsection we show the actual throughput gains achieved with NCQ, while in §6.3 we show the unbearable increase of the latency of interactive applications NCQ causes on a loaded disk. The latency becomes so high to make the system unusable. Accordingly, playing a video is of course just unfeasible. For this reason, we report our results only without NCQ for the video-playback benchmark.

As for the statistics, each benchmark is run ten times and, for each quantity of interest, several aggregated statistics are computed. Especially, for each run and for each quantity, the following values are computed over the samples taken during the run: minimum, maximum, average, standard deviation and 95% confidence interval. The same five statistics are then computed across the averages obtained from each run. We did not find any relevant outlier, hence, for brevity and ease of presentation, we report here only averages across multiple runs (i.e., averages of the averages computed in each run). Finally, hereafter we call just *traces* the information we collected by tracing block-level events (disk request creation, enqueueing, dequeueing, completion and so on) through the Linux *ftrace* facility during experiments.

6.2 Aggregate Throughput

In this benchmark we measure the aggregate disk throughput under four different workloads. Each of these workloads is generated by a given set of file readers or writers starting and executing in parallel, with each file reader/writer exclusively reading/writing from/to a private file. File reads/writes are synchronous/asynchronous and issued back-to-back (greedily). These are the four sets, and the abbreviations we will use to refer to them in the rest of this section: ten *sequential* readers, **10r-seq**; ten *random* readers, **10r-rand**; five *sequential* readers plus five *sequential* writers, **5r5w-seq**; five *random* readers plus five *random* writers, **5r5w-rand**. We denote as *sequential*, or *random*, a reader/writer that greedily reads/writes the file sequentially or at random positions. Each file to read is 5 GB long, or grows up to that size in case of writers.

In the more *sterile* environment used in [1], each file was stored in a distinct disk partition. In this benchmark we put instead all the files in the same partition, in order to get a more realistic scenario. With this configuration, the used filesystems cannot guarantee each file to lie in a single, distinct zone of the disk. Hence even sequential readers may issue a certain fraction of random requests. In addition to the high number of processes that are started and executed in parallel, this lets the workloads in this benchmark be quite demanding for BFQ+ and its budget-assignment rules.

We ran a *long* and a *short* version of the benchmark, differing only in terms of duration: respectively, two minutes and 15 seconds. The purpose of the first version is to assess the *steady-state* aggregate throughput achievable with each scheduler (after removing the samples taken during the first 20 seconds), whereas the second version highlights how quickly each scheduler reaches a high throughput when many applications are started in parallel.

6.2.1 Results without NCQ

As shown in Fig. 2, in the long benchmark both BFQ+ and BFQ achieve an about 24% higher throughput than CFQ with sequential workloads (**10r-seq** and **5r5w-seq**), and are close to the disk peak rate with only sequential readers. As we verified through traces, this good result of BFQ+ and BFQ is mainly due to the fact that the budget-assignment rules let the budgets grow to the maximum allowed value B_{max} (BFQ actually assigns B_{max} even to the initial budgets of the readers and the writers, §4.3). This enables BFQ+ and BFQ to profit by the sequential pattern of each reader for a relatively long time before switching to the next one. In

Disk, size, read peak rate	NCQ-capable	File System	CPU	Distribution
MAXTOR 6L080L0, 82 GB, 55 MB/s	NO	ext3	Athlon 64 3200+	Slackware 13.0
MAXTOR 7L250S0, 250 GB, 61 MB/s	YES	ext3	Pentium 4 3.2GHz	Ubuntu 9.04
MAXTOR STM332061, 320 GB, 89 MB/s	NO	ext4	Athlon 64 3200+	Ubuntu 10.04

Table 1: Hardware and software configurations used in the experiments.

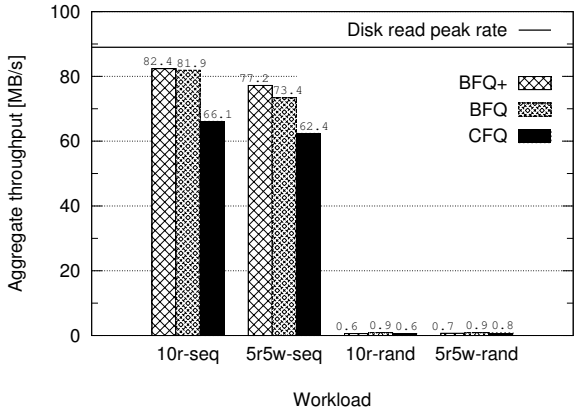


Figure 2: Aggregate throughput achieved, in the long benchmark, by BFQ+, BFQ and CFQ on the third system (without NCQ).

contrast, CFQ switches between processes slightly more frequently, also because its time slice is slightly shorter than the one of BFQ+/BFQ (100 ms against 125). Increasing the time slice would have most certainly improved the performance of CFQ in terms of throughput, but it would have further worsened its latency results (shown in the following subsections). Finally, BFQ+ achieves a slightly higher throughput than BFQ with **5r5w-seq**, mainly because of *H-write-throt* and the fact that sequential writes are *slower* than sequential reads.

As for random workloads, with any of the schedulers the disk throughput unavoidably falls down to a negligible fraction of the peak rate. The performance of BFQ+ with *10r-rand* is however comparable to CFQ, because BFQ+ falls back to a time-slice scheme in case of random workloads (§3.3). The loss of throughput for *5r5w-rand* is instead mainly due to the fact that CFQ happens to privilege writes more than BFQ+ for that workload. And random writes yield a slightly higher throughput than random reads, as could be seen in our complete results. Finally, BFQ achieves a higher throughput than the other two schedulers with both workloads, because it does not throttle writes at all (there is a small percentage of writes, due to metadata updates, also with *10r-rand*). Unfortunately, the little advantage enjoyed in this case is paid with a more important performance degradation in any of the following benchmarks.

As for the short benchmark, BFQ achieves the same aggregate throughput as in Fig. 2 immediately after readers/writers are started. In fact, BFQ assigns them the maximum possible budget B_{max} from the beginning. Though assigning only $B_{max}/2$ as initial budget, BFQ+ reaches however the maximum budget, and hence the highest aggregate throughput, within $1-2$ seconds, thanks to the effectiveness of *H-throughput* and *H-fairness*. CFQ is a little bit slower, and its average aggregate throughput over the first 15 seconds is 59.6 MB/s.

6.2.2 Results with NCQ

The results for the long benchmark with NCQ enabled (on the second system, Table 1) are shown in Fig. 3. BFQ+, BFQ and CFQ have a similar performance with the sequential workloads. This is due to the fact that these schedulers disable disk idling with NCQ, and hence delegate *de facto* most of the scheduling decisions to it. In more detail, the performance of CFQ is moderately worse than BFQ+ and BFQ. As can be verified through traces, it happens because CFQ switches slightly more frequently between processes. This fact causes CFQ to suffer from a more pronounced throughput loss with the random workloads.

As for NOOP (the Linux FIFO disk scheduler), it achieves worse performance than BFQ+ and CFQ with *10r-seq* because it passes requests to the disk device in the same order as it receives them, thus the disk device is more likely to be fed with requests from different processes, and hence driven to perform more seeks. The performance of NOOP improves with *5r5w-seq*, because of the presence of the write requests. In this respect, as can be seen in our complete results, which show also write statistics, NCQ provides a higher performance gain with writes. And—differently from BFQ+, BFQ and CFQ—NOOP does not relegate write requests to a separate single queue that must share the disk throughput with multiple other queues (writes are system-wide and are all inserted in a single queue by BFQ+, BFQ and CFQ). Finally, NOOP provides a 50% performance boost with *5r5w-rand* with respect to BFQ+, because NCQ is even more effective with random write requests. The gist of these results is however that, with NCQ, the service order is actually almost completely out of the control of the schedulers. For this reason the short-benchmark results are quite pointless and for brevity we do not report them here.

6.3 Responsiveness

We measure the start-up time of three common applications of increasing size while one of the four workloads used in §6.2 is being served. According to how *H-low-latency* works, under BFQ+ this quantity is also a measure of the

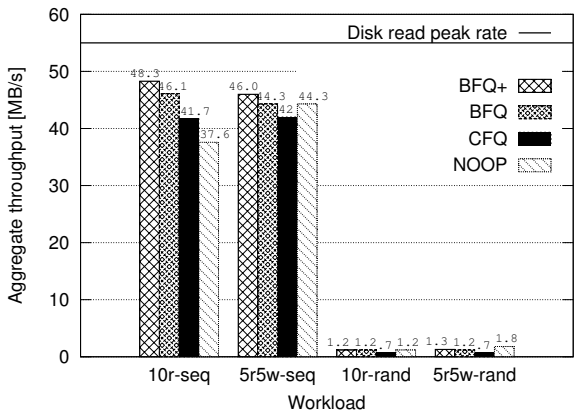
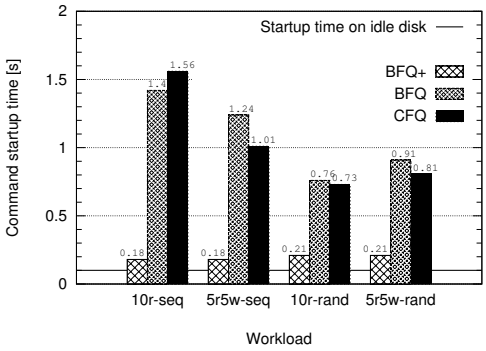


Figure 3: Aggregate throughput achieved, in the long benchmark, by BFQ+, BFQ, CFQ and NOOP (FIFO) on the second system with NCQ.

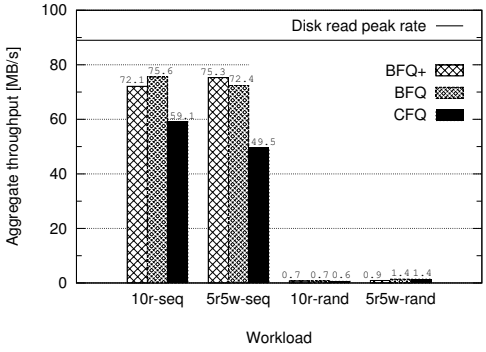
worst-case latency that may be experienced by an interactive application every time it performs some I/O. To get worst-case start-up times we drop caches before invoking each application. The applications are, in increasing size order: *bash*, the Bourne Again shell, *xterm*, the standard terminal emulator for the X Window System, and *konsole*, the terminal emulator for the K Desktop Environment. These applications allow their start-up time to be easily computed. For *bash*, we just let it execute the *exit* built-in command and measure the total execution time. For *xterm* and *konsole*, we measure the time elapsed since their invocation till when they contact the graphical server to have their window rendered. For each run, the application at hand is executed ten times, flushing the cache before each invocation of the application, and with a one-second pause between consecutive invocations. This is the benchmark where the synergy of the heuristics reported in this paper can be best appreciated.

6.3.1 Results without NCQ

Fig. 4(a) shows the *bash* start-up times: under BFQ+ they are up to eight-time lower than under CFQ and BFQ, and quite close to the ones obtained invoking the application on an idle disk. To see how these lower latencies are paid in terms of aggregate throughput, we measured also the latter quantity during the benchmark. As shown in Fig. 4(b), for three out of four workloads, BFQ+ achieves a higher throughput than CFQ too. This lower latency/higher throughput result is due to both *H-low-latency*, and the more accurate scheduling policy of BFQ+ (shared with BFQ). Especially, this policy allows BFQ+ to get low latencies for small-size interactive applications even while assigning high budgets to the readers. As a further proof of this fact, consider that the *bash* start-up time under BFQ+ with *low_latency* disabled was already below 0.55 seconds with any of the workloads (full results in [2]). As can be verified through the traces, without *H-fairness*, BFQ+ could not have achieved such a good result. In fact, BFQ achieves a latency only slightly better than CFQ (Fig. 4(a)) exactly because it lacks this heuristic. The performance of BFQ is even worse than CFQ with *5r5w-seq* and *5r5w-rand*, because BFQ also lacks *W-write-throt*.



(a) *bash* start-up time.



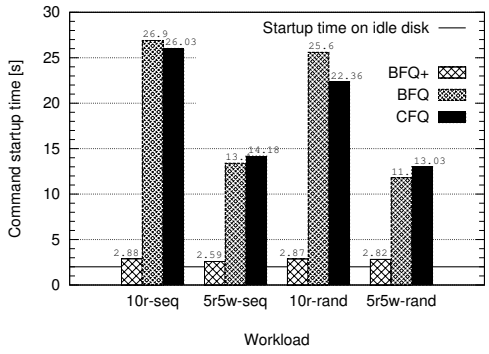
(b) Aggregate throughput.

Figure 4: *bash* start-up times and aggregate throughput during the benchmark (third system without NCQ).

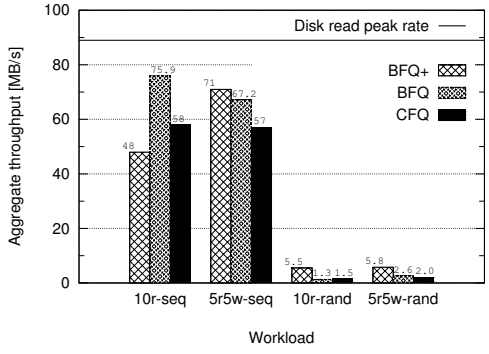
Considering again the throughput, the pattern generated to load an application is usually a mix of sequential and random requests. BFQ+ of course favors this pattern more than CFQ and BFQ with respect to the requests issued by the background workloads. This fact negatively affects the throughput under BFQ+ in case of *throughput-friendly* workloads as, e.g., *10r-seq*. The throughput achieved by BFQ+ is however still much higher than that achieved by CFQ, because the percentage of disk time devoted to *bash* is very low during the benchmark, about 0.20 seconds against a one-second pause between invocations, and because the budget-assignment rules of BFQ+ let the sequential readers get high budgets. As we are about to see, things change when the size of the application increases. Finally, BFQ always achieves a high throughput because it devotes a small percentage of the disk time to *bash*, and assigns high budgets to readers and writers.

For brevity we do not report our results with *xterm*, as these results sit between the ones with *bash* and the ones with *konsole*. Consider then *konsole*, the largest application

of the three. As shown in Fig. 5(a), the latency drop is evident: up to nine times lower start-up time than with CFQ and BFQ. With any workload, BFQ+ is again close to the start-up time achievable on an idle disk. This optimal result is a consequence of the sum of the benefits of the heuristics described in this paper. In particular, adding *H-peak-rate*, *H-throughput*, *H-fairness* and *H-write-throt* alone would let BFQ achieve a start-up time around 20 seconds also with *10r-seq* and *10r-rand*. And it is only thanks to the conjunction of these heuristics and *H-low-latency* that BFQ+ succeeds in achieving the low application start-up times reported in Fig. 5(a).



(a) *konsole* start-up time.



(b) Aggregate throughput.

Figure 5: *konsole* start-up times and aggregate throughput during the benchmark (third system without NCQ).

This low latency is paid with a 20%/36% loss of aggregate throughput with respect to CFQ/BFQ in case of *10r-seq*, as can be seen in Fig. 5(b). It happens because CFQ and BFQ of course favor less than BFQ+ the more-random requests that must be served for loading *konsole*. Differently from *bash*, with *konsole* a significant percentage of time is spent loading the application during the benchmark. On the opposite end, from the full results it could be seen that, though the *konsole*-loading pattern is partially random, favoring it leads however to: 1) a slightly higher throughput than favoring sequential writes, and 2) a definitely higher throughput than favoring purely random read or write requests. For this reason BFQ+ achieves a higher throughput than CFQ and BFQ with the other three workloads.

	bash start-up range [sec]	*konsole* start-up range [sec]
BFQ+	0.27 - 0.32	10.7 - 196
BFQ	0.51 - 0.74	30.9 - 188
CFQ	1.05 - 7.44	14.7 - 2940
NOOP	6.19 - 10.8	1.55 - 408

Table 2: Ranges of start-up times achieved by BFQ+, BFQ, CFQ and NOOP (FIFO) over the four workloads, on the second system with NCQ enabled.

6.3.2 Results with NCQ

With NCQ the results confirm the expected severe degradation of the service guarantees. The variation of the start-up times as a function of the different workloads is so large that they are impossible to clearly represent on charts with linear scale as the ones used so far. Hence we summarize these results in Table 2. For each scheduler and application we report the minimum and maximum (average) start-up times achieved against the four workloads.

As can be seen, BFQ+ still achieves reasonable start-up times for *bash*, whereas *konsole* is now unusable (*xterm* is unusable too). The performance of BFQ+ and BFQ is however better than that of CFQ and NOOP (FIFO), with CFQ taking up to 49 minutes to start *konsole*. There is the outlier of the 1.55 seconds taken by NOOP, precisely with *5r5w-seq*, which we did not investigate further. In general, whereas NOOP does not make any special effort to provide low-latency guarantees, the other three schedulers cannot really be blamed for this bad performance. NCQ basically *amplifies*, in a non-linear way, the latency that would be guaranteed by each of these schedulers without it, because of the two problems discussed in §5. Finally, the results in terms of aggregate throughput match the ones reported in §6.2, hence we do not repeat them.

6.4 Video playback

As already discussed, the litmus test for *H-low-latency* is how this heuristic degrades the latency for non-interactive applications. And soft real-time applications, such as video players, are clearly among the most sensitive ones to the degradation of the guarantees. In this benchmark we count the total number of frames dropped while: 1) a 30-second, medium-resolution, demanding movie clip is being played with the *mplayer* video player, 2) the *bash* command is being invoked with cold caches every 3 seconds (3 seconds is the upper bound to the worst-case start-up time of *bash* with CFQ in this benchmark), and 3) one of the workloads used in §6.2 is being served. *bash* starts to be repeatedly invoked only after 10 seconds since *mplayer* started, so as to make sure that the latter is not taking advantage of any weight

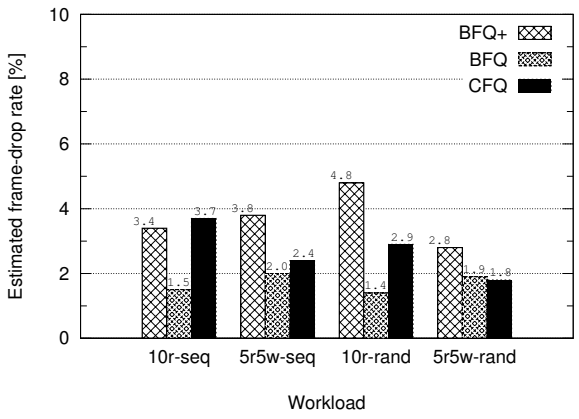


Figure 6: Average frame-drop rate, while the *bash* application is repeatedly invoked and one the four workloads is served.

raising. In contrast, because of its short start-up time, each execution of *bash* enjoys the maximum weight raising and hence causes the maximum possible perturbation.

To show the consequences of the number of frames dropped through a more clear quantity, we computed a conservative estimate of the average frame-drop rate during the last 20 seconds of the playback (the most perturbed ones), assuming a playback rate of 27 frames per second. In this respect, it would have been even more interesting to conduct the *reverse* analysis, i.e., given a well-established frame-drop rate for a high-quality playback, find the most perturbing background workloads for which that threshold is met. To perform such an analysis, we should have taken many variables into account, because the level of perturbation caused by a background workload may depend on many parameters, such as number of readers, number of writers, size of the other applications started during the playback, and frequency at which these applications are started. We did not consider this more complex analysis for lack of space.

Turning back to the actual benchmark we have run, as already said in §6.1, we report here our results on the first system (without NCQ), as this system is the one with the slowest disk. As shown in Fig. 6, the price paid on this system for the low latency guaranteed by BFQ+ to interactive applications is a frame-drop rate not higher than 1.6 times that of CFQ. Note that BFQ exhibits its worse performance with *5r5w-seq* and *5r5w-rand*, mainly because with these workloads it devotes a quite high percentage of the disk time to the write requests. On the contrary, thanks to *H-write-throt*, the relative performance of BFQ+ with respect to CFQ does not get worse under these workloads.

7. Conclusions

In this paper we have described a set of simple heuristics added to the BFQ disk scheduler. We have validated the effectiveness of these heuristics by defining and running, on several heterogeneous systems with single rotational disks, a benchmark suite that mimics real-world tasks. We are currently investigating the issues related to RAIDs and SSDs, together with possible solutions to preserve guarantees also with NCQ. In this respect, we have already devised some improvements for BFQ+ (and integrated them in its last releases [2]). These improvements will be the focus of follow-up work.

References

[1] F. Checconi and P. Valente, "High Throughput Disk Scheduling with Deterministic Guarantees on Bandwidth Distribution", *IEEE Transactions on Computers*, vol. 59, no. 9, May 2010.

[2] [Online]. Available: `http://algo.ing.unimo.it/people/paolo/disk_sched`

[3] [Online]. Available: `http://algo.ing.unimo.it/people/paolo/disk_sched/BFQ-v1-tr.pdf`

[4] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," in *18th ACM SOSP*, Oct. 2001.

[5] A. L. N. Reddy and J. Wyllie, "Disk scheduling in a multimedia I/O system," in *Proc. of MULTIMEDIA '93*, 1993.

[6] L. Reuther and M. Pohlack, "Rotational-position-aware real-time Disk Scheduling Using a Dynamic Active Subset (DAS)," in *Proc. of RTSS '03*, 2003.

[7] A. Molano, K. Juvva, and R. Rajkumar, "Real-time Filesystems. Guaranteeing Timing Constraints for Disk Accesses in RT-Mach," *Proc. of RTSS '97.* , Dec 1997.

[8] A. Povzner et al. "Efficient guaranteed disk request scheduling with fahrrad,"*SIGOPS Oper. Syst. Rev.*, 42, 4, April 2008.

[9] L. Rizzo and P. Valente, "Hybrid: Achieving Deterministic Fairness and High Throughput in Disk Scheduling," in *Proc. of CCCT'04*, 2004.

[10] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk Scheduling with Quality of Service Guarantees," in *Proc. of ICMCS '99.*, 1999.

[11] A. Gulati, A. Merchant, and P. J. Varman, "pclock: an Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, 2007.

[12] W. Jin, J. S. Chase, and J. Kaur, "Interposed Proportional Sharing for a Storage Service Utility," in *Proc. of SIGMETRICS '04/Performance '04*, 2004.

[13] A. Gulati, A. Merchant, M. Uysal, and P. J. Varman, "Efficient and adaptive proportional share I/O scheduling," Hewlett-Packard, Tech. Rep., November 2007.

[14] [Online]. Available: `http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.pdf`

[15] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in *SIGMETRICS '94*, 1994.

[16] R. Geist, J.R. Steele, and J. Westall, "Enhancing Webserver Performance Through the Use of a Drop-in, Statically Optimal, Disk Scheduler", *Proc. 31st Ann. Int. Conf. of the Computer Measurement Group (CMG 2005)*, December 2005.