New version of BFQ, benchmark suite and experimental results

Technical Report

February 11, 2014

This document is a revised and extended version of the paper:

> P. Valente, M. Andreolini, "Improving Application Responsiveness with the BFQ Disk I/O Scheduler", *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12)*. ACM, New York, NY, USA, DOI=10.1145/2367589.2367590 http://doi.acm.org/10.1145/2367589.2367590.

If possible, in a scientific paper please cite the above paper and not (only) this document. Besides, the above paper is, in many parts (including the introduction), crisper and more concise than this document. The main purpose of this document is to provide as much information as possible.

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 document.

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.

As a result of our heuristics: 1) whatever the disk load is, interactive applications are virtually as responsive as if the disk was idle; 2) latencies comparable to CFQ are still guaranteed to time-sensitive, non-interactive applications, as, e.g., audio and video players; 3) a high throughput is achieved also in the presence of many concurrent requests and sudden increases of the workload.

textbfKeywords 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 latter fluctuates. This fraction is established by assigning a fixed weight to each application. In particular, during any time interval in which the set of the applications competing for the disk is constant, the fraction of the throughput guaranteed to each competing application is equal to the ratio between the weight of the application and the sum of the weights of all the competing applications.

BFQ schedules applications as follows. Each application is assigned a budget each time it is scheduled for service. This budget is measured in number of sectors. When selected, an application is granted exclusive access to the disk until either it has consumed all of its assigned budget or it has no more requests to serve. BFQ computes and schedules budgets so as to both achieve a high disk throughput, and closely approximate the service of a perfectly fair *fluid* system. In this ideal system, each application is guaranteed a minimum fraction of the disk throughput, equal to the ratio between the weight of the application and the sum of the weights of the other applications. The rate guaranteed to each application is therefore independent of the size of the budgets assigned to the application. BFQ guarantees that each of the disk requests issued by an application is completed within a bounded delay with respect to when the request would be completed in such an ideal (unfeasible) system.

Consider now applications that perform little I/O periodically or sporadically, and such that, when they need to perform some I/O, a single budget may be enough to serve all of their requests. This is the case for most soft real-time applications (as, e.g., audio and video players), and for non-demanding interactive applications (as, e.g., command-line shells). Consider a batch of requests issued by one of these applications, and suppose that all the requests in this batch is served using a single budget. Finally, consider the delay by which this batch is completed with respect to when the same batch would be completed in the ideal system. The per-request delay guaranteed by BFQ is such that this batch-delay grows with the difference between the budget assigned to the application and the size of the batch. It follows that assigning the smallest possible budget to applications is the key to guarantee them a low per-batch delay.

More precisely, the per-batch delay guaranteed to an application is minimum if, for each batch of requests the application issues, the application is assigned a budget equal to the size of the batch itself. The problem is that, in general, the size of the next batch of requests of an application is not known before the application starts to be served. In fact, an application may issue new requests also while it is being served. In this respect, BFQ computes budgets through a simple feedback-loop algorithm, which does assign small budgets only to applications that issue small batches of requests. This guarantees that these application do enjoy low latencies. It assigns instead large budgets to disk-bound applications, and this is the way

how BFQ also achieves a high disk throughput, as explained in Section 3.

On the opposite end, we stress that, as previously said, BFQ guarantees a bounded per-request delay with respect to an ideal system in which the fraction of the throughput guaranteed to each application during any time interval is independent of the budgets assigned to the application, then also the fraction of the disk throughput guaranteed to each application in the long term by BFQ is independent of the size of the budgets assigned to the application. This property 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, as shown in more detail in Subsection 3.1, BFQ balances this fact: the larger the budgets assigned to an application are, the less frequently BFQ lets the application access the disk.

BFQ has been implemented in the Linux kernel (by Fabio Checconi and Paolo Valente) and compared against several research and production schedulers [1]. It outperformed those schedulers in fulfilling the requirements of applications ranging from file transfer (fairness and high throughput) to WEB and video streaming servers (low latency).

1.1 Limitations of the original version of BFQ

Thanks to the above properties, BFQ allows a user to enjoy the smooth playback of a movie while downloading other files, and/or while some service is accessing the disk. Yet, a further critical feature is needed for a high-quality user experience: *responsiveness*, i.e., a low latency in loading and hence starting up applications, and in completing the batches of I/O requests that interactive applications issue sporadically. Actually, starting applications (invoking commands) is one of the most common actions a user performs.

What about responsiveness under BFQ? Consider a typical scenario in which all applications are automatically assigned the same weight by the operating system. According to the same, above-discussed properties, a small-size application does enjoy a low start-up time, because serving a small batch of requests is enough to load the application (as confirmed also by our experimental results in Subsec. 7.3). But what happens if larger applications are started on a loaded disk? This time the effects of the fairness of BFQ are reversed: if an application is guaranteed only a fraction of the disk throughput and other applications are competing for the disk as well, then transferring many sectors may take a long time for that application. For example, if an application takes 2 seconds to start on an idle disk, then the same application takes about 20 seconds (on average) under BFQ, if ten files are being read at the same time. This is an unbearable wait for a user.

A further important limitation 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 of BFQ should be verified across heterogeneous systems, including also RAIDs and solid-state drives (SSD). More demanding workloads should be considered too, i.e., workloads generated by even more processes and where the request patterns may vary suddenly over time (switching, e.g., from non-disk-bound to disk-bound).

Finally, modern rotational and solid-state devices can internally queue disk requests and serve them either in the best order for boosting the throughput, or even concurrently. As of now, Native Command Queueing (NCQ) is probably the most established technology. In previous work [1] an important problem has been highlighted theoretically: with most mainstream applications, letting a device freely prefetch and reorder requests should cause both fairness and latency guarantees to be violated with any scheduler, including BFQ. However, experimental evidence was still missing.

1.2 Contributions reported in this document

In this document we report the outcome of our first step in overcoming the limitations of BFQ. Especially, we describe the following contributions.

- A set of simple heuristics added to BFQ, with the following three goals: to improve responsiveness, to preserve a high throughput under demanding workloads and to improve the robustness with respect to heterogeneous disk devices. The resulting new ver-

sion of BFQ is called BFQ+ in this document and BFQ-v1 in the patchsets that introduce BFQ in the Linux kernel [2].

- 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 results collected by running the above suite with BFQ+, BFQ and CFQ, on three Linux systems with a single rotational disk. The systems differed in both hardware and software configurations. One system was NCQ-capable, and we ran the suite also with a simple 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 reasons are the following. As for the former class of schedulers, on a loaded disk it is hard for them to guarantee a low latency to interactive applications. In contrast, many real-time and proportional-share research schedulers do provide latency guarantees comparable to BFQ. But, as shown in [1], they may suffer from low throughput and degradation of the service guarantees with mainstream applications. Addressing these issues again is out of the scope of this document. More details on the issues related to all these classes of schedulers can be found in Section 5.

Our results with BFQ+ can be summarized as follows: differently from CFQ and whatever the disk load is, 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 these applications receive more than their fair share of the disk throughput. Nevertheless, the heuristic fits the accurate service provided by BFQ well enough to still guarantee that non-interactive, time-sensitive applications (as, 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 (Section 3). This fact and, above all, the new low-latency features described in this document 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]. Especially, when the disk is busy serving some disk-bound application, the latency of interactive applications may become so high to make the system unusable.

The other important systems to consider are RAIDs and SSDs. We are investigating the issues related to these systems, together with further improvements for SSDs and possible solutions to preserve guarantees also with NCQ. We have already devised some improvements for BFQ (and integrated them in the last releases of the scheduler [2]). These improvements are built on top of the heuristics that we show in this document. Results with RAIDs and SSDs will then be the focus of follow-up work.

Organization of this document

In Section 2 we introduce both the system model and the common definitions used in the rest of the document. The original version of BFQ is then described in Section 3, while the proposed heuristics can be found in Section 4. Finally, after describing the related work and introducing the problems caused by NCQ in Section 5, we describe the benchmark suite in Section 6 and report our experimental results in Section 7.

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.

At the opposite end, 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 *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.

3 The original BFQ algorithm

In this section we outline the original BFQ algorithm (see [1] for full details). BFQ+ is identical to BFQ, apart from that it also contains the heuristics described in Sec. 4. To make the main steps clear, we first describe a simplified version of the algorithm, with some features omitted or just sketched. Then we provide more details in the following subsections.

The logical scheme of BFQ is depicted in Fig. 1. Solid arrows represent the paths followed by the requests until they reach the disk device. There is a request queue for each application, where the latter inserts its requests by invoking the interface *add_request()* function (details on the Local C-LOOK scheduler below). 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.

Disk access is granted to one application at a time, denoted as the *active application*. Each application has a *budget* assigned to it, measured in number of disk sectors. When an application becomes the active one, it is served exclusively until either this budget is exhausted

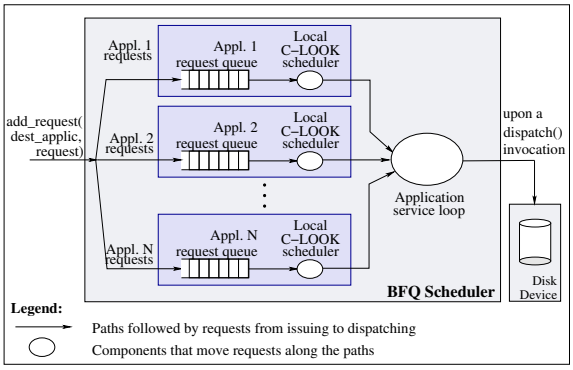


Figure 1: Logical scheme of BFQ.

or the backlog of the application empties (the application becomes idle). Then BFQ selects the new active application, and so on. In other words, the active application cannot be preempted until one of the above two events occur. More in detail, each time BFQ turns from not having any application backlogged to having at least one application backlogged, the *Application service loop* depicted in Fig. 1 starts. This loop, repeated until there is at least one backlogged application, can be sketched as follows:

1. **Choice of the next active application** The internal fair-queueing scheduler, called B-WF^{2}Q+ and described in some detail in Subsec. 3.1, chooses the next active application among the backlogged ones.

2. **Request dispatch** The loop blocks, waiting for the disk device to invoke the *dispatch()* function. When there is at least one backlogged queue, operating system mechanisms around BFQ guarantee that the disk 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 active application; this request is extracted from that queue and dispatched to the disk (right side of Fig. 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 (the actual condition used here is slightly more complex, as detailed in Subsecs. 3.4 and 3.5) jump to step 3, otherwise repeat step 2.

3. **Application deactivation and budget recomputation** The application stops being the active one, and it is assigned a new budget. A simple feedback-loop algorithm, described in detail in Subsec. 3.3, is used to compute the new budget.

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

Under BFQ each application is associated with a *weight*, which controls the fraction of the throughput guaranteed to the application. If an application is not assigned a weight explicitly, then BFQ sets the weight of the application to a common, system-wide value. The description of the internal B-WF^{2}Q+ scheduler provided in this subsection also shows in which way the weight of an application determines the fraction of the throughput received by the application.

The internal B-WF^{2}Q+ scheduler is the key element that enables BFQ to closely approximate the service that would be guaranteed to each application by an ideal, perfectly fair fluid system [1]. Though achieving this goal with a few simple operations, B-WF^{2}Q+ is based on non-trivial concepts. Especially, B-WF^{2}Q+ is a slightly extended version of the WF^{2}Q+ packet scheduler [3], adapted to disk scheduling and integrated with a special *timestamp back-shifting* rule to preserve guarantees in presence of synchronous requests. In this subsection we do not enter into timestamp details, and just outline the scheduling policy of B-WF^{2}Q+ and its properties; see [1] for a complete description of the algorithm.

The definition of B-WF^{2}Q+ is based on the concept of *corresponding fluid* system. First, by corresponding we mean that this system serves the same applications as the real system, i.e., the system on which BFQ runs, and that, at all times, it has the same total throughput as the real system. By fluid we mean instead that this system can serve more than an application at a time. Especially, the perfectly fair fluid system considered so far serves all the backlogged applications at the same time, and provides each with a fraction of the total throughput equal to the weight of the application divided by the sum of the weights of the applications backlogged at the same time instant. B-WF^{2}Q+ internally simulates a corresponding fluid system that, for efficiency issues (see [1]), is slightly less accurate than a perfectly fair one. In more detail, the difference with respect to the perfectly fair system is that this less accurate system may not serve some of the backlogged applications during some time intervals. On the bright side, simulating this system is computationally less expensive than simulating a perfectly fair one. For ease of presentation, hereafter we pretend that also this system provides a perfectly fair service. In any case, the worst-case throughput distribution and delay guarantees reported below are correct.

B-WF^{2}Q+ implements the following policy: when asked for the next active application at step 1 of the application service loop, it considers only the applications whose current budget would have already started to be served in the simulated fluid system, and, among these, it returns the one whose budget is the next to finish (in the fluid system). Of course B-WF^{2}Q+ is an on-line algorithm, and hence does not know the future, so the next budget to finish is computed assuming optimistically that: 1) no further application becomes backlogged before this budget is finished in the fluid system, and 2) the application does not empty its backlog before consuming all of this budget. As long as these two conditions are met, B-WF^{2}Q+ succeeds in letting BFQ finish budgets in the same order as the fluid system; otherwise some budgets may be served out of order.

As proved in [1], this scheduling policy allows BFQ to provide the following optimal worst-case guarantees for a non-preemptible budget-by-budget service scheme: BFQ guarantees that each request is completed with the minimum possible worst-case delay with respect to when the request would be completed in the fluid system in the worst case (more precisely BFQ guarantees the minimum possible worst-case delay for a budget-by-budget service scheme). In addition, BFQ guarantees to each application and over any time interval, the minimum possible lag with respect to the minimum amount of service that the fluid system guarantees to the application during the same time interval. With *worst-case* request completion times in the fluid system and with *minimum* amount of service guar-

anteed to the application in the fluid system, we mean the completion times and the amount of service guaranteed in the fluid system in case all applications are backlogged.

In more detail, both the worst-case delay of the requests of an application and the worst-case lag of the application are upper-bounded by a quantity equal to the sum of: 1) a component proportional to the maximum budget that BFQ may assign to applications, and 2) a component proportional to the maximum possible difference between the budget that may be assigned to the application and the number of sectors that the application *consumes* before it may become idle. To cancel or at least reduce the second component, BFQ should assign a *tight* budget to each application. This is one of the purposes of the budget-assignment algorithm described in Subsection 3.3.

Before concluding this subsection, it is worth stressing the following fact: B-WF^{2}Q+ guarantees that either delays or lags cannot accumulate over time (see the proofs in [1] for details). And, as previously said, in the fluid system the fraction of the disk throughput guaranteed to each application is *determined only by the weight* of the application (divided by the sum of the weights of the other backlogged applications). This fraction is then *independent of the size of the budgets* assigned to the application, as the concept of budget is not used at all in the definition of the service provided by the fluid system. In the end, since B-WF^{2}Q+ guarantees a bounded delay and lag with respect to the fluid system, then, in the long term, also 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 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 by postponing the service of an application in proportion to the budget currently assigned to the application. In fact, the larger the budget assigned to an application is, the later this budget is completed in the fluid system that B-WF^{2}Q+ approximates. Finally, as shown in Subsection 3.3, also the budgets assigned to an application are *independent of the weight* of the application.

For the reader interested at least into the intuition on how B-WF^{2}Q+ achieves a bounded deviation from the fluid system, in the next subsection we sketch, intuitively, the proof of the delay guarantees provided by B-WF^{2}Q+. Reading this subsection is not necessary to understand the rest of this document.

3.2 Sketch of the proof of the delay guarantees

We can intuitively assess the worst-case request completion times guaranteed by B-WF^{2}Q+ by considering the following two alternatives for the start time of a budget. The first is that the start of the budget is not delayed by any out-of-order service. In this case it is easy to prove that the budget, and hence any request served using it, is finished no later than when it would have been finished in the fluid system. The second is that there is an out-of-order service. According to what previously said, this may happen only if one or both the above assumptions made by B-WF^{2}Q+ do not hold. In particular:

- An application becomes backlogged and its budget happens to have a lower finish time, in the fluid system, than the one of the budget of the currently active application. In this case, as proven in [1], the budget of the newly backlogged application will be completed with a delay, with respect to the fluid system, not higher than the time needed to finish (in the real system) the out-of-order budget already under service.

- The active application becomes idle, i.e., empties its backlog, before consuming all of its budget. This case has the worst consequences on delay. Once activated, the application does finish its requests sooner than expected, but, exactly for this reason, its activation may have been postponed for *too long*. In fact, to decide the service order, B-WF^{2}Q+ computes the expected finish time of the budget of each application in the simulated fluid system, assuming that the application consumes *all* of its budget once become active. Hence, if an application A becomes idle before the budget is exhausted, it has been assigned a too high finish time. As a consequence, other applications may be unjustly activated, and have their budgets served, before A. As proven in [1], the worst-case delay with respect to the fluid system in this case is equal to the time to serve the portion of the budget that the application does not use (i.e., the dif-

ference between the budget and the number of sectors served before the application becomes idle), at a speed equal only to the fraction of the disk throughput guaranteed by the fluid system to the application.

Of course, if both the above scenarios occur, the two delay components sum to each other. Using the same arguments it is possible to prove the guarantees of B-WF^{2}Q+ in terms of lag.

3.3 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.

In the first place, BFQ might just assign a small budget to any application. This would minimize both components of the upper bound to the delay mentioned in Subsec. 3.1. The problem is that small budgets would cause BFQ to frequently switch between applications, and to not fully benefit from the many sequential accesses that could be performed by serving each of the possible sequential applications for a sufficiently long time. In this respect, consider that, 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. After this random access, in the best case, 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. 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.

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 Subsecs. 4.2 and 4.3, where we show how, by enhancing these components, we improve the service properties and increase the throughput under BFQ.

3.4 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 applications 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 disk* 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 disk as it arrives. Though apparently counterintuitive, on rotational devices this wait usually results in a boost of the disk throughput [4] with sequential and synchronous applications. In this respect, note that most mainstream applications issue synchronous requests. As shown in [1], disk 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.

3.5 Preserving fairness with random requests

BFQ imposes a time constraint on disk usage: once obtained access to the disk, the application under service must finish either its budget or its backlog within T_{max} time units (T_{max} is the maximum time slice defined in Subsec. 3.3), otherwise a *budget timeout* fires, and the application is deactivated.

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 it has used only part of it. This causes B-WF^{2}Q+ to assign a higher finish time to the next budget of the application, and ultimately reduces the frequency at which applications incurring budget timeouts can access the disk. On the other hand, also sequential applications may run into a budget timeout. In Subsec. 4.4 we propose a refinement of the budget-overcharging strategy for this case, which allows both a higher aggregate throughput and a lower latency to be achieved.

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.

In some of the following subsections we use the phrase "detected as random". In this respect, BFQ computes the average distance between the requests of an application using a low-pass filter, and deems the application random if this distance is above a given threshold (currently 8192 sectors).

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 let the application be loaded 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. 2, 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

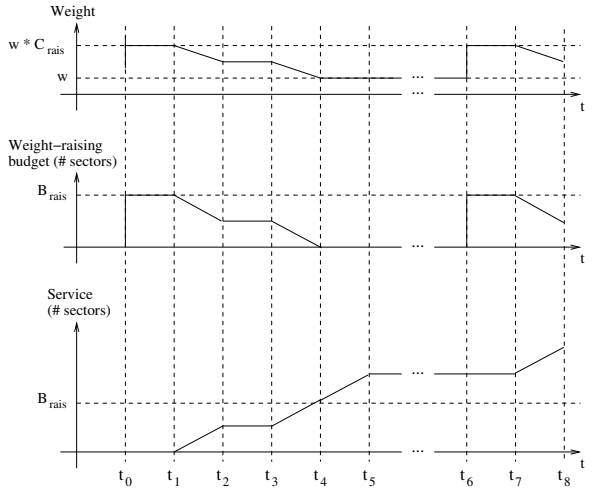


Figure 2: Weight raising for an application created at time t_0, become idle at time t_5 and again backlogged at time t_6; the application is served only during $[t_1, t_2]$, $[t_3, t_5]$ and $[t_7, t_8]$.

the weight, which is described below. The bottommost graph shows instead 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. 2, 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 said, in [1] it is shown that disk idling is instrumental in preserving service guarantees in the presence of deceptive idleness. 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, the latter 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 tem-

porary 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 reasonable 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, Subsec. 4.3).

4.2 A new peak rate estimator

As showed in Subsec. 3.3, 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 (Subsec. 3.5). Besides, a too high value of B_{max} degrades service properties unnecessarily (Subsec. 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 (Subsec. 3.3), 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

To describe the following set of simple heuristics, we need first to add a detail: when the active application is deactivated for whichever reason, BFQ, and hence BFQ+, also control whether it has been too *slow*, i.e., it has consumed its last-assigned budget at such a low rate that it would have been impossible to consume all of it within the maximum time slice T_{max} (Subsec. 3.5). In this case, in BFQ, the application is always (over)charged the full budget to reduce its disk utilization, exactly as it happens with random applications (Subsec. 3.5).

We found the two situations below, occurring frequently, in which this behavior causes throughput loss, increased latencies or even both. We also report a third situation, related to temporarily random applications, that gives rise to similar problems. For each situation, we report the new behavior of BFQ+ (the constants are again the result of our tuning with different disks).

1. 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. For this reason, if a slow application is deactivated after receiving very little service (at most $1/8$ of the maximum budget), it is not charged the full budget.

2. Due to ZBR, an application may be deemed slow when it is performing I/O on the slowest zones of the disk. However, unless the application is really *too* slow, not reducing its disk utilization is more profitable in terms of disk throughput than the opposite. For this reason an application is never charged the full budget if it has consumed at least a significant part of it $(2/3)$.

3. We have seen that some applications generate really few and small, yet very far, random requests at the beginning of a new disk-bound phase. The large distance between these initial requests causes the average distance (computed using a low-pass filter as stated at the beginning of this section) to remain high for a non-negligible time, even if then the application issues only sequential requests. Hence, for a while, the application is unavoidably detected as random. As a consequence, before the following modification, the disk-idling timeout was set to very low values if the application issued synchronous requests (Subsec. 3.4), and this often caused loss of disk throughput and increased latency. Now the disk-idling timeout for a random application can be set to a very low value only after the application has consumed at least a minimum fraction $(1/8)$ of the maximum budget B_{max}.

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 schedulers [5, 6, 7, 8, 9]; 2) proportional-share timestamp-based schedulers [10, 11, 12, 13]; and 3) proportional-share round-robin schedulers [14, 15]. In the next subsection we focus on the first two classes of schedulers and discuss an important issue related them. Then, in Subsec. 5.2, we provide more details on round-robin schedulers. Here we note that most of the schedulers in any of the three classes combine their main policies with some effective algorithm to achieve a high disk throughput. Examples of the latter algorithms are SCAN (Elevator), C-SCAN, LOOK, C-LOOK [16] and Anticipatory [4]. It is important to highlight that, depending on the application, some of these throughput-boosting algorithms can achieve, on their own, latencies comparable or lower than a scheduler aimed at providing a predictable service. Unfortunately, this is not the case for interactive applications, as a throughput-boosting algorithm may not serve them for a long time if, e.g., some sequential access is being performed in parallel. In contrast, in [17] Anticipatory is shown, for example, to provide low latencies with a WEB-server workload. For such a workload, the authors also show that *Table-Building Bus Driver*, a scheduler aimed at minimizing request response times, achieves a little less than half the latencies of a fine-tuned scheduler as CFQ. This is about the same result achieved by BFQ for this type of workloads according to [1].

Argon [18], Cello [19], APEX [20] and PRISM [21] are frameworks for providing QoS guarantees. It is worth mentioning also Real-Time Database Systems (RTDBS), which are architectures for performing database operations with real-time constraints [22]. As for the Argon storage server, its goal is to provide each service with at least a configured fraction of the throughput it achieves when it has the server to itself, within its share of the server. This result is achieved with such mechanisms as request prefetching and cache partitioning. The other above-mentioned architectures provide instead for the use of one or more underlying disk schedulers. Hence their

overall service guarantees depend on the properties of the adopted schedulers. In this respect, as already said, in the next subsection we discuss an important issue related to existing real-time and proportional-share timestamp-based schedulers. Finally, consider that any scheduler may have to interact with the internal queueing performed by a modern disk drive. The issues related to disk-drive internal queueing are described in Subsection 5.3.

5.1 Real-time and timestamp-based proportional-share schedulers

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 theoretically and experimentally shown in [1]. In fact, the service guarantees of these schedulers do 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. In [1] it is also shown that the original properties of some of these schedulers can be partially or completely recovered with some simple extensions. Since these issues are out of the scope of this document, we do not repeat here the detailed discussion and the experiments reported in [1].

5.2 Round-robin schedulers

CFQ is a production-quality round-robin disk scheduler, fine-tuned over the years and currently under active development. As such, it is probably one of the best-performing schedulers in its class. For this reason, we use it as a reference to describe the main properties of round-robin schedulers. BFQ owes to CFQ the idea of exclusively serving each application for a while. But, differently from BFQ, CFQ grants disk access to each application for a fixed time slice (as BFQ basically does only for random applications, Subsec. 3.5). Slices are scheduled in a round-robin fashion. 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 (shown in detail in Subsec. 3.1). Finally, a *low_latency* tunable has been recently added to CFQ: when enabled, CFQ tries to reduce the latency of interactive applications in a similar vein as BFQ.

5.3 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 its only pending request is dispatched. If the disk is not idled and other applications are backlogged, any scheduler would of course serve another application each time the application A becomes idle. As a consequence, the application A would not obtain the high share of the disk throughput (or disk time) it should receive.

6 The benchmark suite

In this section we describe the purpose and the motivations of the suite of benchmarks through which we measure the performance of the schedulers in the next section. The reader not interested into these aspects may skip this section and move directly to the experimental results in the next section.

We wanted to be able to assess the performance of a scheduler in terms of relevant figures of merit for real-world tasks. To this purpose, we focused on the following performance indexes and, for each quantity, we consider the following worst-case scenarios.

Aggregate disk throughput As general scenarios for measuring disk throughput we can consider the reading and the writing of multiple files at the same time, with each file accessed sequentially or randomly (i.e., with read/write requests scattered at random positions within each file). The resulting set of workloads represents both a typical disk I/O micro-benchmark, and a worst-case (heavy-load) scenario for file download, upload and sharing. Hereafter, we refer to workloads like these when we say heavy workloads. Given the obvious importance of disk throughput as a figure of merit, we measure it also in all of the next scenarios.

Responsiveness We focus primarily on the latency experienced by a user in one of the most common actions she/he performs: starting applications (invoking commands). As a worst-case scenario, we consider the *start-up* time of applications—i.e., how long it takes for applications to start doing their job from when they are launched—with cold caches and in presence of additional heavy workloads. According to how the low-latency heuristic described in Subsec. 4.1 works, under BFQ this quantity is in general a measure of the worst-case latency experienced by an interactive application every time it performs some I/O.

Latency of soft real-time applications As already discussed, the litmus test for the weight-raising heuristic added to BFQ is how it degrades the quality of the service provided to non-interactive applications. And soft real-time applications are clearly among the most sensitive ones to the degradation of the guarantees. A video player is an important representative of soft real-time applications for desktop or handheld systems. To assess the quality of the service it experiences, we can count the number of frames dropped during the playback of a movie clip, when, in parallel, a command is repeatedly invoked with cold caches, and in presence of additional heavy workloads.

Throughput guaranteed to code-development applications Some of the fundamental applications in this category are compilers and revision control systems. With respect to these applications, a key figure of merit for a programmer is most certainly how fast a compilation or checkout/merge from a repository advance in presence of other heavy workloads.

Fairness As for fairness, extensive experiments have been carried out in [1], and we verified that those results are preserved also in this new version. Hence for brevity in the next section we do not provide either these results or details about the fairness benchmark contained in the suite.

Database workloads As shown in detail in [1], with these workloads any scheduler achieves a disk throughput equal to a negligible fraction of the peak rate, and other solutions, as, e.g., caching, should be adopted for better performance. We do not repeat these results in the next section and, as of now, the suite doe not contain any database benchmark.

We analyzed several benchmark tools to find some suitable to evaluate the first four figures of merit. Unfortunately, we found appropriate tools only for measuring aggregate disk throughput. Many of these tools are also aimed at measuring single-request latency under workloads made of sequential or random requests issued back-to-back (*greedy* readers/writers). These patterns match only partially the ones generated by soft real-time or interactive applications. The former applications usually issue small batches of requests spaced by a short, often fluctuating, time period. The latter may instead generate a mix of sequential and random synchronous requests, issued *almost* back-to-back, i.e., spaced by a short but variable idle time (as we also saw in our traces). As a further

consequence of these discrepancies, the workloads generated by these benchmark tools lack the heterogeneity of request patterns that can be found in a real system. In the end, as we verified experimentally, the actual latency experienced by interactive and soft real-time applications may differ with respect to the one that could be inferred from the statistics reported by these tools.

In view of this situation, we implemented an *ad hoc* benchmark suite that covers the above scenarios and collects the desired throughput and latency measures. Especially, every benchmark mimics, or is actually made of, real-world I/O tasks. The suite is publicly available [2]. Each of the benchmarks mentioned in this section is described in detail in the next section (apart from the fairness benchmark), together with our results. New benchmarks may however have been added to the suite after we have written this document.

7 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 document. The experiments themselves are then reported in the following subsections. These experiments concern aggregate throughput, responsiveness, latency for soft real-time applications and throughput guaranteed to code-development applications. Especially, the latency guaranteed to soft real-time applications 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 Subsecs. 4.1–4.5: *H-low-latency*, *H-peak-rate*, *H-throughput*, *H-fairness* and *H-write-throt*.

We performed the experiments reported in the following subsections using the benchmark suite introduced in the previous section. We describe each benchmark in detail at the beginning of each subsection. All the results and statistics omitted in this document can be found in [2]. In addition, the benchmark suite contains the general script that we used for executing the experiments reported in this document (all these experiments can then be repeated easily).

7.1 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 (Subsec. 3.3). 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 (Subsec. 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

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.

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 Subsec. 7.3 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.

7.2 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.

7.2.1 Results without NCQ

As shown in Fig. 3, 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, Subsec. 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 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 and 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 (Subsec. 3.5). 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. 3 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.

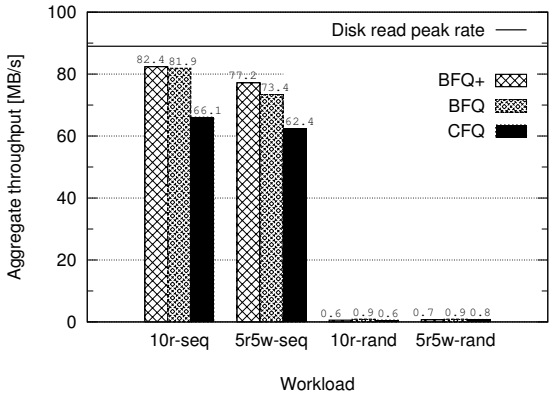


Figure 3: Aggregate throughput achieved, in the long benchmark, by BFQ+, BFQ and CFQ on the third system (without NCQ).

7.2.2 Results with NCQ

The results for the long benchmark with NCQ enabled (on the second system, Table 1) are shown in Fig. 4. 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 mul-

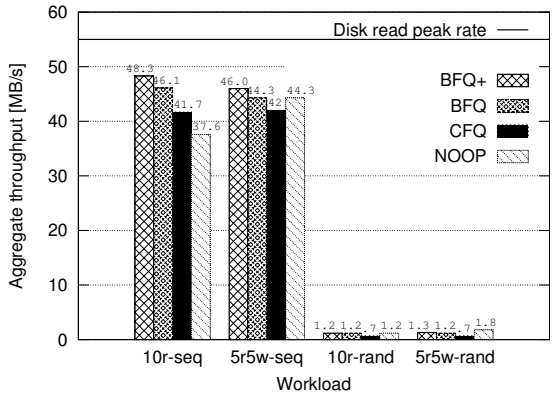


Figure 4: Aggregate throughput achieved, in the long benchmark, by BFQ+, BFQ, CFQ and NOOP (FIFO) on the second system with NCQ.

tiple 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.

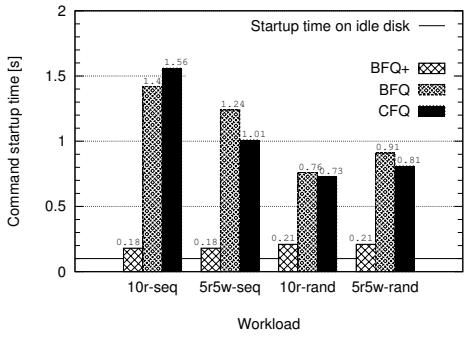
7.3 Responsiveness

We measure the start-up time of three common applications of increasing size while one of the four workloads used in Subsec. 7.2 is being served. 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 document can be best appreciated.

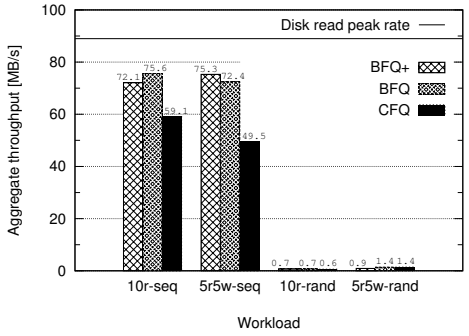
7.3.1 Results without NCQ

Fig. 5(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. 5(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. 5(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*.

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



(a) *bash* start-up time.

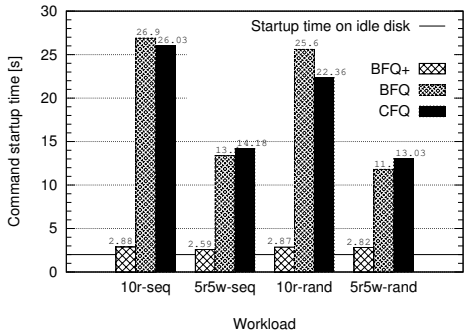


(b) Aggregate throughput.

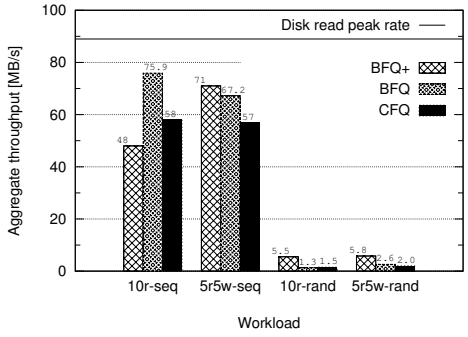
Figure 5: *bash* start-up times and aggregate throughput during the benchmark (third system without NCQ).

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. 6(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 document. 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. 6(a).



(a) *konsole* start-up time.



(b) Aggregate throughput.

Figure 6: *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. 6(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+

	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.

achieves a higher throughput than CFQ and BFQ with the other three workloads.

7.3.2 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 Sec. 5. Finally, the results in terms of aggregate throughput match the ones reported in Subsec. 7.2, hence we do not repeat them.

7.4 Video playback

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 Subsec. 7.2 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 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 do not consider this more complex analysis for the moment.

Turning back to the actual benchmark we have run, as already said in Subsec. 7.1, 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. 7, 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.

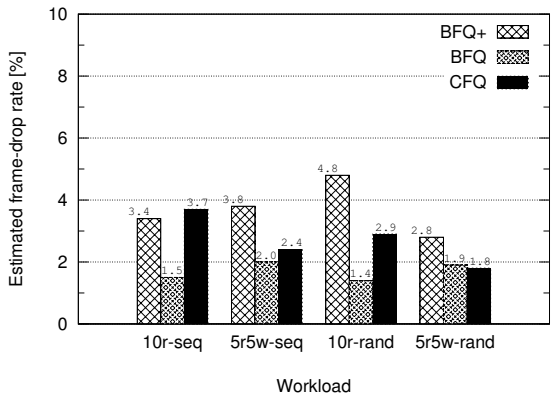


Figure 7: Average frame-drop rate, while the *bash* application is repeatedly invoked and one the four workloads is served.

7.5 Code-development applications

In this benchmark we measure the progress of three different code-development tasks (started with cold caches): the compilation of the Linux kernel, the *checkout* of a branch in a clone of the Linux *git* repository, and the *merge* of two branches. One of the four workloads used in Subsec. 7.2 is served in parallel with the task at hand. Each of these tasks has an initial phase during which the number of disk requests it issues is quite unpredictable and may vary a lot across different runs (see the suite for details [2]). Hence, we record the progress of these tasks during two minutes since this phase is finished.

A significant part of the work done by the three tasks is writing files, but writes are system-wide and hence end up in a common flush queue in BFQ+, BFQ and CFQ. As a consequence, none of these schedulers has the opportunity to save the writes generated by these tasks from being overwhelmed by the ones of the greedy writers. For this reason, the *5r5w-seq* and *5r5w-rand* workloads cause the three tasks to almost completely starve. We report here only the results with the other two workloads, generated only by readers.

As for kernel compilation, we verified through tracing that really few read requests are issued. These requests are quickly served with both BFQ+ and CFQ, whereas the application spends most of the time either using the CPU or waiting for its write requests to be handed over to the virtual memory subsystem. In the end, most of the (little) control of BFQ+ and CFQ over the progress of a task like this is related to how the scheduler balances writes with respect to the reads generated by the background workloads. And BFQ+ balances them more or less like CFQ. Consequently, the progress of the compilation is about the same with both schedulers, as shown in Fig. 8(a). In contrast, since it lacks *H-fairness*, BFQ delays the sporadic read requests issued during the compilation more than BFQ+ and CFQ. This worsens its performance. The results in terms of throughput are instead virtually the same as the ones for *10r-seq* and *10r-rand* in Fig. 5(b): BFQ+ and BFQ achieve again a higher throughput than CFQ with *10r-seq*.

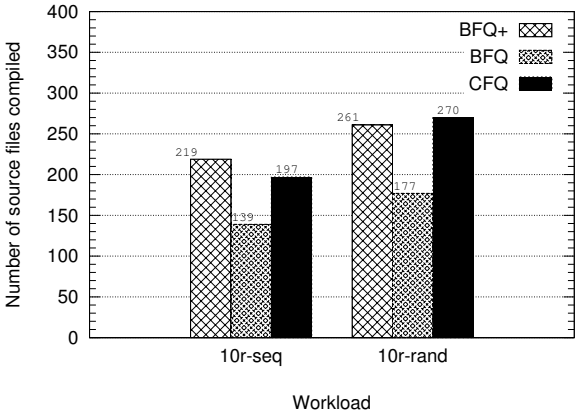


Figure 8: Kernel compilation progress, measured in number of source files compiled during the benchmark.

As for *checkout* and *merge*, Fig. 9 groups together the results for both tasks, reporting, for each task, the progress against each workload. Differently from the kernel compilation, BFQ+ and BFQ achieve better results than CFQ with these two tasks. The main reason is that these tasks are more read-intensive than a compilation. Especially, they allow BFQ+ to take full advantage of *H-fairness* and to definitely outperform BFQ. Regarding aggregate throughput the results are again very close to the ones for *10r-seq* and *10r-rand* in Fig. 5(b).

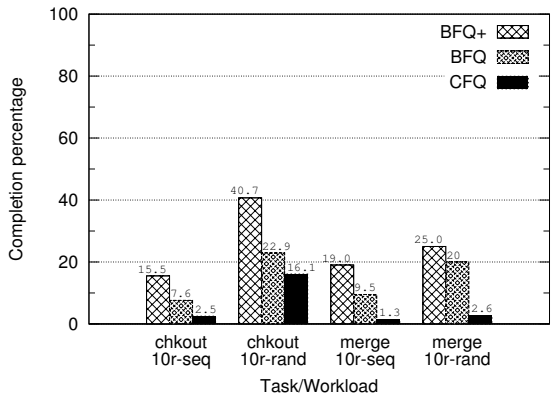


Figure 9: Progress of *checkout* and *merge* tasks, measured in percentage of the task completed during the benchmark.

8 Conclusions

In this document we have described a set of simple heuristics added to the original BFQ disk scheduler. The resulting new scheduler is named BFQ-v1 in the patchsets that introduce BFQ in the Linux kernel [2]. These heuristics are aimed at improving responsiveness and robustness across heterogeneous devices, as well as achieving high throughput under demanding workloads. We have validated the effectiveness of these heuristics by running, on several heterogeneous systems with single rotational disks, a benchmark suite that mimics real-world tasks. The next step will be to tackle RAIDs and SSDs, and to further investigate ways for preserving guarantees also with NCQ.

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://algogroup.unimore.it/people/paolo/disk_sched`

[3] J. C. R. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, 1997.

[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] S. Daigle and J. Strosnider, "Disk Scheduling for Multimedia Data Streams,", in *Proc. of the IS&T/SPIE*, 1994.

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

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

[8] 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.

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

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

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

[12] 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.

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

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

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

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

[17] 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.

[18] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance insulation for shared storage servers," in *In Proc. of the 5th USENIX Conference on File and Storage Technologies*, 2007.

[19] P. J. Shenoy and H. M. Vin, "Cello: a disk scheduling framework for next generation operating systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, 1998.

[20] T. P. K. Lund, V. Goebel, "APEX: adaptive disk scheduling framework with QoS support," *Multimedia Systems*, vol. 11, no. 1, 2005.

[21] A. L. N. Reddy, J. Wyllie, and K. B. R. Wijayaratne, "Disk scheduling in a multimedia I/O system," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 1, no. 1, 2005.

[22] B. Kao and H. Garcia-Molina, "An overview of real-time database systems,"*Advances in Real-Time Systems*, 1995.