High Throughput Disk Scheduling with Fair Bandwidth Distribution

Paolo Valente and Fabio Checconi

Abstract—Mainstream applications—such as file copy/transfer, Web, DBMS, or video streaming—typically issue synchronous disk requests. As shown in this paper, this fact may cause work-conserving schedulers to fail both to enforce guarantees and provide a high disk throughput. A high throughput can be, however, recovered by just idling the disk for a short time interval after the completion of each request. In contrast, guarantees may still be violated by existing time-stamp-based schedulers because of the rules they use to tag requests. Budget Fair Queuing (BFQ), the new disk scheduler presented in this paper, is an example of how disk idling, combined with proper *back-shifting* of request time stamps, may allow a time-stamp-based disk scheduler to preserve both guarantees and a high throughput. Under BFQ, each application is always guaranteed—over any time interval and independently of whether it issues synchronous requests—a bounded lag with respect to its reserved fraction of the total number of bytes transferred by the disk device. We show the single-disk performance of our implementation of BFQ in the Linux kernel through experiments with real and emulated mainstream applications.

Index Terms—Scheduling, secondary storage, quality of service.

1 Introduction

Most mainstream applications, such as file transfer, Web, DBMS, Video on Demand, or Internet TV, require moving data to/from disk devices. Meeting the disk bandwidth and per-request delay requirements of these applications and at the same time achieving a high throughput is not an easy task. The first problem is that the time needed to serve a request once dispatched to the disk device, hereafter called *service time*, is highly variable. Causes are seek and rotational latencies, variation of the transfer rate with the sector position, caching.[1] In addition, few, if any, of the existing controllers export these physical parameters, which makes it difficult to predict service times based on request positions on the disk. Finally, most mainstream applications usually issue one request or batch of requests at a time. Especially, they *block* until the only outstanding request/batch has been completed. We denote this type of request or batch of requests as *synchronous*, for it can be issued only after the outstanding batch/request has been completed. The peculiar arrival pattern of synchronous requests may cause guarantee and disk throughput problems.

Guarantee violations may occur with time-stamp-based schedulers, i.e., schedulers that time-stamp requests as a function of their arrival time and in essence dispatch them to the disk in ascending time stamp order. The root of the problem is that the arrival of a synchronous request may be arbitrarily delayed by a scheduler by just delaying the dispatching of the preceding request. Then a delayed synchronous request may get a higher time stamp with respect to the one it would have got if not delayed. This higher time stamp may finally let the request wait for the service of more requests before being dispatched to the disk. Unfortunately, delaying the service, and hence, the completion of the request will delay the arrival of the successive synchronous request of the same application, and so on. In the end, by just delaying the service of its requests, the scheduler may force the application to issue requests at a *deceptively* lower rate. If this anomaly occurs, the scheduler just fails to guarantee the reserved bandwidth or the assigned request completion times to the application (even to a greedy one).

In addition, a minimum amount of time is needed for an application to handle a completed request and to submit the next synchronous one. On one hand, this fact may further contribute to violating guarantees with time-stamp-based schedulers. On the other hand, it may also prevent a work-conserving scheduler from achieving a high disk throughput. From the disk device standpoint, an application is *deceptively idle* until it issues the next synchronous request [1]. During one such idle time, the disk head may be moved away from the current position by a work-conserving scheduler, thus losing the chance of a close access for applications performing mostly sequential IO. The problem is mitigated by the fact that operating systems typically perform *read-ahead* for request patterns deemed sequential.

Overprovisioning would be a way to easily guarantee a predictable and short request service time without dealing with the above problems. Unfortunately, it entails high purchase, powering and cooling costs [2], and purposely wastes disk bandwidth. In fact, the only option to improve the disk utilization and meet the requirements of many competing applications is properly scheduling disk requests. Several schedulers—such as SCAN (Elevator), C-SCAN, LOOK, or C-LOOK [3]—have been defined to achieve the first goal. These algorithms are often implemented also inside modern

1. Other less influential sources of variability are sector sparing and dynamic variation of the disk parameters due to thermal variations.

• P. Valente is with Università di Modena e Reggio Emilia, Via Vignolese 905/b, 41100 Modena, Italy. E-mail: paolo.valente@unimore.it.
• F. Checconi is with the ReTis Lab., Scuola Superiore S.Anna, Via Moruzzi 1, 56124 Pisa, Italy. E-mail: fabio@gandalf.sssup.it.

Manuscript received 12 Nov. 2008; revised 16 June 2009; accepted 13 Nov. 2009; published online 12 May 2010.
Recommended for acceptance by S.H. Son.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-11-0573.
Digital Object Identifier no. 10.1109/TC.2010.105.

disk devices, which can internally queue requests and service them in the best order to boost the throughput. Finally, to achieve a high throughput also in the presence of deceptive idleness, most Linux standard disk schedulers extend these policies with *disk idling*: they do not dispatch any other request to the disk for a short time interval (in the order of the seek and rotational latencies) after a synchronous request has been completed. By doing so, they give a chance to the (possible) next request of the same application to arrive before the disk arm is moved away. The Anticipatory (AS) disk scheduler [1] extends C-LOOK in this sense.

However, to meet the requirements of most types of applications, guarantees must also be provided on bandwidth distribution or request completion times. In this respect, the problem of algorithms aimed only at maximizing the disk throughput is that in the worst case, they may delay the service of a request until the whole disk has been read or written (age-based policies, as in AS, can be added to mitigate the problem). In contrast, many schedulers, e.g., SCAN-EDF [4], SATF-DAS [5], JIT [6], Hybrid [7], YFQ [8], pClock [9], adaptive DRR [10], CFQ [11], and adaptations of SFQ [12], have been proposed to provide control on request completion time and/or on bandwidth distribution, while at the same time trying to keep the throughput high.

Apart from CFQ, all of these schedulers are work-conserving and none of them takes the delayed arrival problem into account. Hence, as previously discussed, they may provide a low disk throughput, and if time-stamp-based, may violate guarantees. For similar reasons, guarantees for synchronous requests are violated with any scheduler if the disk device performs internal queuing, as discussed in more detail in Section 4. Our experimental results with a single disk thoroughly confirm the expected loss of guarantees for all the analyzed work-conserving time-stamp-based schedulers, as well as a loss of disk throughput in case of applications performing mostly sequential accesses. Of course, on the opposite end, adopting a naïve disk idling approach in a multidisk system, i.e., idling the entire array of disks on the completion of each sequential synchronous request, may cause throughput loss [13].[2] Some more effective strategies are mentioned together with our proposal in the next section.

In contrast, a nontime-stamp-based scheduler should not suffer from either problem provided that some mechanism is adopted to handle deceptive idleness. This is the case for round-robin schedulers, e.g., CFQ, which does perform disk idling. Unfortunately, as discussed in Section 4 and experimentally shown in Section 5, they exhibit a higher delay/jitter in request completion times than the time-stamp-based scheduler we propose.

1.1 Proposed Solution

In this paper, we propose a *proportional share* time-stamp-based disk scheduler, called Budget Fair Queuing (BFQ). As in any proportional share scheduler, in BFQ, each application is guaranteed its reserved fraction (share) of the disk throughput, irrespective of its behavior. Hence, neither the request arrival pattern of the application needs to be known nor the application itself needs to be modified to provide such a guarantee.

BFQ serves applications as follows: First, when enqueued, an application is assigned a *budget*, measured in number of sectors to transfer. Once selected, the application gets exclusive access to the disk. During the service of the application, disk idling is performed to wait for the arrival of synchronous requests (only if these requests are deemed sequential, see Section 2.2). Finally, if the application runs out of either its backlog or its budget, it is deselected and assigned a new budget. This service scheme allows BFQ to achieve a high throughput if applications issue mostly sequential requests. In contrast, for random workloads, it is not as optimal as a global policy, e.g., C-LOOK. It is, however, worth noting that with random workloads, even an optimal policy can achieve only a small fraction of the disk rate. In fact, where possible, caches are usually tuned so as to reduce disk access and achieve feasible response times.

Applications are scheduled by the internal Budget-$\mathrm{WF^2Q+}$ ($\mathrm{BWF^2Q+}$) scheduler as a function of their budgets. The latter is a slightly extended version of $\mathrm{WF^2Q+}$ [14], a proportional share (fair-queuing) packet scheduler. B-$\mathrm{WF^2Q+}$ basically differs from the original packet scheduler in that, first, it handles the case where an application becomes idle before consuming all of its budget, and second, it properly *shifts backward* time stamps to conceal delayed arrivals. Due to this characteristic and to the fact that it works in the service and not in the time domain, BFQ provides the following guarantee to each application, over *any* time interval, with any workload and regardless of the disk physical parameters: each application is guaranteed the minimum possible *lag*, achievable by serving applications budget by budget, with respect to the minimum amount of service, measured in number of sectors transferred, that the application should receive according to its reserved share of the disk throughput and to the total amount of service provided by the system. A loose upper bound to this lag is $3B_{max}$, where B_{max} is the maximum budget that can be assigned to any application. In general, BFQ can be fine-tuned to achieve the desired trade-off between throughput boosting and maximum per-application lag, by tuning B_{max} and a few other configuration parameters. In addition, a simplified interface is provided for users not concerned with low-level details.

The above *sector* guarantees can be turned into *time* guarantees as follows: First, the aggregate throughput achieved with the desired values of the configuration parameters must be measured for the expected worst-case request pattern (see Section 3.4). Then, worst-case time guarantees can be computed as a function of the aggregate throughput with a simple closed-form expression. Note that, on one side, no other disk physical parameter needs to be known or measured, whereas, on the other side, there is no possibility with any scheduler to provide practical time guarantees without knowing at some degree at least the locality of the expected request pattern.

With regard to computational cost and implementation issues, BFQ is defined without using disk physical parameters, and has $O(logN)$ worst-case cost, in the number of competing applications, per request insertion or dispatch (this cost boils down to $O(1)$ if approximate implementations [15] of B-$\mathrm{WF^2Q+}$ are adopted). We have implemented

2. In [13], an extensive comparative analysis of the performance of several disk schedulers is reported, in terms of aggregate throughput, with different workloads and also in presence of internal queuing and multiple disks.

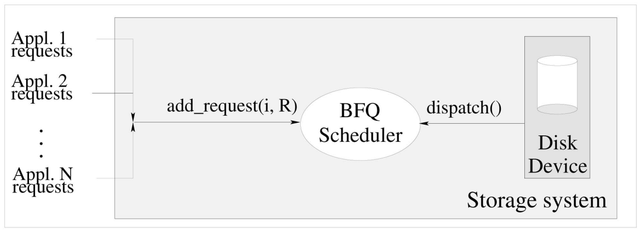

Fig. 1. System model.

BFQ in the Linux kernel (currently for 2.6.21-29 kernels [16]) and experimentally evaluated its single-disk performance with file transfer, Web server, DBMS, and video streaming applications (Section 5). In general, the main contribution of this paper is showing through BFQ a way to combine disk idling and time stamp back-shifting to preserve both guarantees and a high throughput in the presence of synchronous requests. This technique may be applied also to other time-stamp-based schedulers. Finally, to preserve a high throughput in a multiple-disk system, the disk idling scheme described so far should be extended to allow, e.g., one outstanding request per disk. Or alternatively, a hierarchical approach might be adopted, leaving the disk idling task only to the local per-disk schedulers. Careful investigation of these solutions is out of the scope of this paper.

1.2 Organization of the Paper

In Section 2, we describe BFQ and report its computational cost, whereas in Section 3, we show its service properties in the sector and time domains. In Section 4, we provide a brief survey of related work. Finally, in Section 5, we compare the performance of BFQ against several research and production schedulers.

2 BFQ

After defining the system model in the next section, we introduce the logical scheme and the main algorithm in the successive two sections, where B-WF^{2}Q+ is used as a black box providing application enqueue/dequeue operations. We then describe B-WF^{2}Q+ in detail in Sections 2.4 and 2.5.

2.1 System Model

We consider a *storage system* composed of a disk device and the BFQ scheduler, as in Fig. 1. The former contains a disk, which we model as a sequence of contiguous fixed-size sectors, each identified by its *position* in the sequence. The disk device services two types of *disk requests*, respectively, the reading and the writing of a set of contiguous sectors. After receiving the *start command* or completing a request, the disk device asks for the next request to serve by invoking the function `dispatch` exported by the BFQ scheduler.

At the opposite end, requests are issued by the N applications served by the storage system (applications here stand for the possible entities that can compete for disk access in a real system, e.g., *threads* or *processes*). The meaning of the notations hereafter introduced is also summarized in Table 1. The ith application issues its jth request R_i^j by invoking the function `add_request` exported by the scheduler, and passing the index i and the request R_i^j. We define as *size* L_i^j of R_i^j the number of sectors to read or write,

TABLE 1
Definitions

Symbol	Meaning
R_i^j	j-th request issued by the i-th application
L_i^j	Size of R_i^j
L_{max}	$\max_{i,j} L_i^j$
a_i^j, s_i^j, c_i^j	Arrival, start and completion time of R_i^j
$W_i(t)$	Amount of service received by the i-th application
$W(t)$	Total amount of service delivered by the system
T_{wait}	Time waited before deeming an application as idle
$B_{i,max}$	Maximum budget assigned to the i-th application
B_{max}	$\max_i B_{i,max}$
$f(t_1, t_2)$	$f(t_2) - f(t_1)$
ϕ_i	Weight of the i-th application
T_{FIFO}	Queueing time after which (queued) requests must be served in FIFO order

and as *position/end* of R_i^j the position of the first/last of these sectors. We say that two requests are *sequential* if the position of the second request is just after the end of the first one. We define as *arrival*, *start*, and *completion* time of a request the time instants a_i^j, s_i^j, and c_i^j at which the request R_i^j is issued by the ith application, starts to be served, and is completely served by the disk device, respectively. We say that a request is *synchronous* if it can be issued by an application only after the completion of its previous request. Otherwise, the request is denoted by *asynchronous*.

We say that an application is *receiving service* from the storage system if one of its requests is currently being served. Both the amount of service $W_i(t)$ received by an application and the total amount of service $W(t)$ delivered by the storage system are measured in number of sectors transferred during $[0, t]$. For each application i, we let $B_{i,max}$ denote the dynamically configurable maximum budget, in number of sectors, that BFQ can assign to it. We define $B_{max} \equiv \max_i B_{i,max}$.

We say that an application is *backlogged* if it has pending requests. In addition, to deal with the delayed arrival and the deceptive idleness problems, an application is denoted by *quasi-backlogged* at time t if either it is backlogged or it is not backlogged but its backlog emptied not before time $t - T_{wait}$, where T_{wait} is a system-wide dynamically configurable time interval. Otherwise, the application is deemed as *idle*. Moreover, we say that an application i enjoys the *short-or-independent arrival property* (SI property for short) if, for each request R_i^j, either R_i^j is asynchronous or $a_i^j - c_i^{j-1} \leq T_{wait}$. The actual adherence of real-world applications to the SI property is discussed in Section 3, whereas hereafter, we assume that applications do enjoy this property.

Each application has a fixed weight ϕ_i assigned to it. Without losing generality, we assume that $\sum_{q=1}^{N} \phi_q \leq 1$. Given a generic function of time $f(t)$, we define $f(t_1^-) \equiv \lim_{t \to t_1^-} f(t)$ and $f(t_1, t_2) \equiv f(t_2) - f(t_1)$. Finally, given any time interval $[t_1, t_2]$ during which the ith application is continuously quasi-backlogged, we define its *reserved service* during $[t_1, t_2]$ as $\phi_i \cdot W(t_1, t_2)$. Suppose that R_i^{j+1} is a synchronous request, and let $\overline{c}_i^j$ be the time instant at which the request R_i^j would be completed if the ith application received exactly its reserved service during $[a_i^j, \overline{c}_i^j]$. We say that the arrival of R_i^{j+1} is (deceptively) *delayed* if $c_i^j > \overline{c}_i^j$.

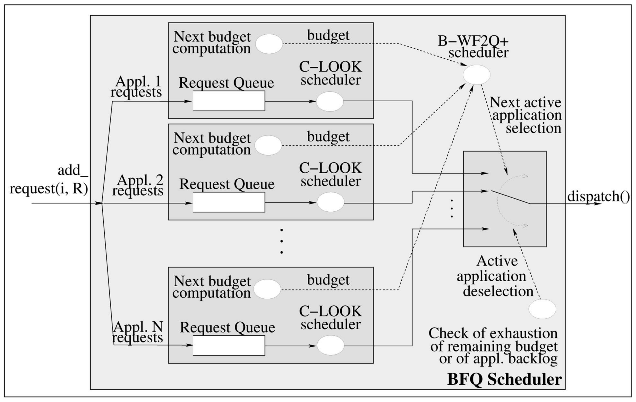

Fig. 2. BFQ logical scheme.

2.2 Logical Scheme

The logical scheme of BFQ is depicted in Fig. 2. Different from Fig. 1, here, solid arrows represent the paths followed by requests until they reach the disk device. On the contrary, dashed arrows represent flows of information or internal commands. Finally, circles represent algorithms or operations. There is a *request queue* for each application. Requests are inserted into the right queue by the `add_request` function.

At any time, each application has a *budget* assigned to it, measured in number of sectors. Let B_i^l be the lth budget assigned to the ith application. At system start-up, all applications are assigned the same *default* budget B_i^0. Disk access is granted to one application at a time, denoted by the *active application*. When the new active application is selected, its current budget is assigned to a special *remaining budget* counter. Each time a request of the active application is dispatched, the remaining budget counter is decreased by the size of the request. Moreover, if the request queue gets empty at time t, a timer is set to $t + T_{wait}$ to wait for the possible arrival of the next request. This may leave the disk idle, but prevents BFQ from switching to a different application if the active application is deceptively idle and enjoys the SI property. In addition to not breaking a possible sequence of close or sequential accesses, this waiting is also instrumental in concealing delayed arrivals, as shown in Section 2.5. However, waiting for the arrival of nonsequential requests provides no benefit. Hence, as CFQ, BFQ automatically reduces T_{wait} to a very low (configurable) value for applications performing random IO (currently 2 ms, see the code for details [16]). If the active application issues no new request before the timer expiration, it is deemed idle. The active application is exclusively served until either there is not enough remaining budget to serve the next request or the application becomes idle. At this point, the next budget B_i^{l+1} of the application is computed. Basically, B_i^{l+1} is obtained by increasing or decreasing B_i^l, depending on whether the application consumed all of its budget or ran out of its backlog before consuming it. The next active application is then chosen by B-WF^{2}Q+, which schedules enqueued applications as a function of their budgets (see Section 2.5).

To guarantee a controllable per-application maximum queuing time, the order in which the requests are extracted from the queue of the active application depends on a user

```
active_appl = none ; // reference to the currently active application
remaining_budget = 0 ; // remaining budget of the active appl.

// input: application index, request issued by the application
add_request(int i, request R) {
  appl = applications[i] ; // reference to the i-th application

  // insert R in the application queue
  enqueue(R, appl.queue) ;

  if (appl.queue.size == 1) { // the queue was empty
    if (appl != active_appl)
      b–wf2q+_insert(appl) ; // Sec.2.5
    else  // appl is the active application
      if (waiting_for_next_req()) // deceptive idleness
        unset_timer() ; // next req arrived: stop waiting
  }
}

request dispatch() { // output: request to serve
  if (all_applic_are_idle() OR waiting_for_next_req())
    return no_request ;

  if (active_appl != none AND
      remaining_budget <
      C–LOOK_next_req(active_appl.queue).size) {
        b–wf2q+_update_vfintime(active_appl,
            active_appl.budget – remaining_budget) ;

        if (active_appl.budget + BUDG_INC_STEP <=
            active_appl.max_budget)
            active_appl.budget += BUDG_INC_STEP ;
        else
            active_appl.budget = active_appl.max_budget;

        b–wf2q+_insert(active_appl) ;
        active_appl = none ;
  }
  if (active_appl == none) {
    // get and extract the next active appl. from b-wf2q+ (Sec. 2.5)
    active_appl = b–wf2q+_get_next_application() ;
    remaining_budget = active_appl.budget ;
  }
  // get and remove the next request from the queue of the active application
  next_request = dequeue_next_req(active_appl.queue) ;

  // for simplicity at this point we assume that next_request.size ≤
  // remaining_budget, see the code [16] for details
  remaining_budget –= next_request.size ;
  if (is_empty(active_appl.queue))
      set_timer(T_wait) ; // start waiting for the next request

  // account for this service in b-wf2q+
  b–wf2q+_inc_tot_service(next_request.size) ;

  return next_request ;
}

timer_expiration() { // called on timer expiration
  active_appl.budget =
    active_appl.budget – remaining_budget ;
  b–wf2q+_update_vfintime(active_appl,
            active_appl.budget) ; // Sec. 2.5
  active_appl = none ; // no more the active application
  // dispatch() will take care of selecting the next active application
}
```

Fig. 3. BFQ main algorithm.

configurable T_{FIFO} parameter. When the next request of the ith application is to be dispatched at time t, if $a_i^j + T_{FIFO} > t$ holds for all the queued requests R_i^j, then the next request is chosen in C-LOOK order [3]; otherwise, the oldest request is picked.

2.3 Main Algorithm

The main BFQ algorithm is shown in Fig. 3 using pseudocode. The function `add_request` first inserts the new request R in the application queue, then, if there are more than one request, nothing is to be done. Otherwise, if the application is not the active one, the application is enqueued in the B-WF^{2}Q+ scheduler. If the (active) application is waiting for the arrival of the next request, the timer is unset.

The function `dispatch` returns a *no request* indication if all applications are idle or if the active application is waiting for the arrival of the next request (disk idling). On the contrary, if the active application has not enough remaining budget for its next request, the application is deselected (lines 24-38). Moreover, through the call to the function `b-wf2q+_update_vfintime` (described in the next section) at lines 27 and 28, the application time stamps are updated to account only for the service received. The fact that the application did not empty all of its backlog is assumed as an indication that it issues batches of requests with a larger cumulative size than the last assigned budget. Hence, the application budget is increased by a configurable quantity provided that the resulting value is not higher than $B_{i,max}$ (lines 30-34). Finally, the application is enqueued into B-WF^{2}Q+ with its new budget (line 36). Then, if no application is active (lines 39-43), the next active application is picked (and removed) from the B-WF^{2}Q+ scheduler. The next request R to serve is extracted from the queue of the active application and the remaining budget counter is decremented (lines 44-49).

Finally, if the queue of the active application becomes empty, the timer is set to the current time plus T_{wait} (lines 50 and 51). If no new request is issued by the application before timer expiration, the function `timer_expiration` (lines 59-66) gets called: the application is declared idle, and a new budget equal to the previously consumed one is assigned to it. Finally, the time stamps of the application are properly updated to account only for the actual amount of service it received (see the next section).

The adopted simple linear increase/instantaneous decrease of budgets is of course only one of the possible options. It showed good performance in terms of both aggregate throughput and short-term guarantees in our experiments. For simplicity, many low-level details have been omitted here, as the fact that disk idling is obviously performed only for synchronous requests. The interested reader is referred to [16]. Before leaving this section, it is worth noting that BFQ actually provides a flexible framework in that both the B-WF^{2}Q+ and C-LOOK schedulers can be replaced with different schedulers (Fig. 2). Hence, different trade-offs among type of guarantees, computational cost, and throughput boosting degree can be achieved.

2.4 Disk Weighted Fair Queuing

In this section, we provide a brief survey of the main concepts behind the original WF^{2}Q+ algorithm (see [14] and [17] for details), (re)formulated in the disk scheduling domain. These concepts are the basis for B-WF^{2}Q+, which is then described in detail in the next section.

Hereafter, we use the term *batch* to denote the set of the requests served using a given budget. We define a batch as pending at time t if it has not yet been completely served at time t (each application has at most one pending batch at a time). We define two systems as *corresponding* if they serve the same applications, and at any time instant, they provide the same total amount of service per time unit.

WF^{2}Q+ approximates on a packet by packet basis the ideal service provided by a work-conserving packet-based *fluid system*. In B-WF^{2}Q+, we map the concept of packet into the concept of batch: B-WF^{2}Q+ approximates on a batch by batch basis the service provided by a corresponding fluid system that may serve more than one application at a time. Hereafter, this system is called just the ideal system, as opposed to the real system, i.e., to the storage system in Fig. 1. The ideal system is used as a reference because it distributes the total service among applications so as to guarantee to each application a bounded lag, over any time interval, with respect to its reserved service (see [18] for details). Moreover, in the ideal system, each request is completed no later than the time instant at which it has to be completed according to the reserved service of the issuing application. On the contrary, as shown in Section 3, this does not hold in the real system. Hence, a synchronous request in the real system may arrive later than in the ideal system. This fact may cause request arrivals to be deceptively delayed in the real system.

An application is *eligible* at time t if its pending batch would have already started to be served in the ideal system at time t. B-WF^{2}Q+ tries to complete the service of application batches in the same order as the ideal system. Moreover, it chooses the next application to serve only among the eligible ones.

This policy is efficiently implemented by time-stamping each application with the values assumed by a special *application virtual time* function at the start and completion times of its pending batch in the ideal system. These two values are called *virtual start* and *finish time*, $S_i(t)$ and $F_i(t)$, of the application or, equivalently, of its pending batch at time t. They are computed as a function of a common *system virtual time* function $V(t)$ when an application is (re)inserted into the scheduler (see [14] or [17]). Although it seems a contradiction in terms, the unit of measure of the virtual time is the service (see [18] for details). Especially, the virtual time of each backlogged application grows as the amount of service received by the application, divided by its weight.

2.5 B-WF^{2}Q+ and Time Stamp Back-Shifting

The B-WF^{2}Q+ algorithm is shown in Fig. 4 using pseudocode. An application is inserted into the scheduler by calling the function `b-wf2q+_insert`. If the application is not the active one, then, since T_{wait} is waited for before deactivating an application and applications enjoy the SI property, `b-wf2q+_insert` is called as a consequence of the arrival of an asynchronous request. In this case, the virtual start and finish times of the application are computed as a function of the actual time at which `b-wf2q+_insert` is called, i.e., the actual request arrival time, using the same formulas as in WF^{2}Q+ [14].

On the other hand, if the application to insert is the active one, then according to Fig. 3, line 36, the application is necessarily being enqueued into B-WF^{2}Q+ after a deactivation and not because of the arrival of a request. Let R_i^j be the next request to serve of the application (there is certainly one). As explained below in detail, after a deactivation, F_i is assigned the value assumed by the application virtual time upon the completion of the last served request, in this case R_i^{j-1}, in the ideal system (Fig. 3 lines 27 and 28). Let $\overline{F_i}$ be this

```
 1 V = 0 ; // system virtual time
 2
 3 // input: the application to insert
 4 b-wf2q+_insert(appl) {
 5   if (appl != active_appl) // then an asynchronous req arrived
 6     appl.S = max(V, appl.F) ; // use actual arrival time
 7   else // in this case the active appl. is being deactivated
 8     // to preserve guarantees in case of delayed arrival, timestamp
 9     // the application as if the next request to serve arrived at the same
10     // time as the last request of the just terminated batch
11     appl.S = appl.F ;
12   appl.F = appl.S + appl.budget / appl.weight ;
13   // add the application to the internal data structure
14   set_of_enqueued_appl.insert(appl) ;
15 }
16
17 // extract and return the next application to serve
18 b-wf2q+_get_next_application() {
19   if (there_is_no_eligible_appl())
20     // certainly there are enqueued applications, otherwise this
21     // function does not get called (see function dispatch)
22     V = set_of_enqueued_appl.minS() ;
23   next_appl = eligible_appl_with_minF() ;
24   set_of_enqueued_appl.remove(next_appl) ;
25   return next_appl ;
26 }
27
28 // called upon each request dispatch
29 b-wf2q+_inc_tot_service(service) {
30   V = V + service ;
31 }
32
33 // update application virtual finish time
34 b-wf2q+_update_vfintime(appl, received_service) {
35   appl.F = appl.S + received_service / appl.weight ;
36 }
```

Fig. 4. B-WF^{2}Q+.

value. According to [14], the exact value of the application virtual time upon the arrival of R_i^j should have been equal to $\max(\overline{F_i}, V(a_i^j))$. On the contrary, in B-WF^{2}Q+, $\overline{F_i}$ is unconditionally assigned to S_i, as if R_i^j had arrived at a time instant between the arrival time a_i^{j-1} and the completion time in the ideal system of the previous request R_i^{j-1}. This fictitious backward shift conceals the possibly delayed arrival of R_i^j and of the successive requests served using the same budget as R_i^j. Moreover, the worst-case guarantees of the other applications are not endangered, as the guarantees provided to each application are independent of the arrival time of the requests of the other applications.

The function `b-wf2q+_get_next_application` returns the eligible application with the minimum virtual finish time and removes it from the internal data structure (lines 23-25). In case there are quasi-backlogged applications, but no one is eligible (lines 19-22), $V(t)$ is *pushed up* to the minimum virtual start time among *all* the quasi-backlogged applications (the latter coincides with the applications enqueued in B-WF^{2}Q+ when `b-wf2q+_get_next_application` is invoked). Since an application is eligible if and only if its virtual start time is not higher than the system virtual time, this jump guarantees B-WF^{2}Q+ to be work-conserving [14]. However, as shown in Section 2.2, the overall BFQ algorithm is not work-conserving, as it waits for the arrival of a new request before serving the next one.

Pushing up the system virtual time is a delicate operation with respect to the fictitious backward shift of arrival times. Let the ith application be one of the applications that are idle in the real system at time $\bar{t}$ when a jump is performed. Suppose for a moment that a delayed synchronous request R_i^j arrives after time $\bar{t}$. Since the ith application is not taken into account in computing $V(\bar{t})$, the jump would be conceptually incompatible with a fictitious backward shift of the arrival of R_i^j to (or before) time $\bar{t}$. Hence, it is easy to show that it would not be possible to conceal this delayed arrival without violating the guarantees of the ith application. Fortunately, if the ith application enjoys the SI property, it is not possible that $a_i^j > \bar{t}$ because T_{wait} seconds are waited for before invoking `b-wf2q+_get_next_application`.

$V(t)$ is also incremented by the size of the just dispatched request upon each request dispatch (function `b-wf2q+_inc_tot_service` at lines 29-31). For a perfect tracking of the ideal system, $V(t)$ should be continuously increased by the amount of service provided by the disk device. Of course, this is impossible in a real system because the disk device does not export continuous information on the amount of service provided. Suppose that an idle application issues a new request R while a request $\bar{R}$ is under service, and let $\bar{c}$ be the completion time of $\bar{R}$. Due to the stepwise increment of $V(t)$, the application may be time-stamped as if R actually arrived at time $\bar{c}$. Of course, since the application is enqueued before time $\bar{t}$, the worst-case effect of this wrong time-stamping is delaying the service of R as if it arrived at time $\bar{t}$. The consequences of this fact on the service guarantees are shown in Section 3.

The last important difference between B-WF^{2}Q+ and WF^{2}Q+ is that the latter also handles the fact that an application may not use all of its budget. This possibility affects only time guarantees, as shown in Section 3.2. Here, we highlight just that, before an application that did not use all of its budget may be enqueued again, its virtual finish time is properly updated by calling `b-wf2q+_update_vfintime` (lines 34-36) to account only for the actual service received.

Finally, B-WF^{2}Q+ can be implemented at $O(\log N)$ or $O(1)$ cost per application insertion/extraction [15], depending on whether exact or approximate time stamps are used. Since all the other operations in Fig. 3 have $O(1)$ cost, the overall BFQ scheduler can be implemented at $O(\log N)$ or $O(1)$ cost per request insertion/extraction.

The basic algorithm reported in this section does not contain many of the details of the complete version. The latter, e.g., autonomously adapts to a dynamic application set, and to allow the users to choose the weights in a simpler and more flexible way, it poses no constraint on the values of the weights [19]. Of course, in this case, it may happen that $\Phi_{TOT} \equiv \sum_{i=1}^{N} \phi_i > 1$, but the service properties of BFQ still hold after replacing ϕ_i with $\frac{\phi_i}{\Phi_{TOT}}$ in the following inequalities.

3 Service Properties

In this section, we report the service properties of BFQ, in both sector (bandwidth distribution) and time domains. We also show how to perform admission control and provide time guarantees. Finally, we show how to achieve the desired trade-off between fairness granularity and throughput boosting.

Before proceeding, it is important to identify the set of applications for which the service properties of BFQ actually hold also in the presence of delayed arrivals. From Section 2, we know that BFQ conceals the delayed arrivals of the requests issued by the applications that meet the SI property. Hence, BFQ guarantees to these applications the same amount of service and the same per-request completion time as if the arrival of each of their synchronous requests would not have been delayed. As a consequence, one would set T_{wait} as high as possible to include as many applications as possible. However, the value of T_{wait} has an important impact on the disk throughput. It may provide significant boosting in the presence of deceptive idleness, but only if its value is in the order of the seek and rotational latencies [1], namely a few milliseconds. In contrast, higher values may cause progressive performance degradation, as the disk may be left idle for too long.

Applications commonly alternate phases during which they make intense use of the disk device and phases during which they rarely access it. Fortunately, even for the abovementioned beneficial low value of T_{wait}, during the former phases, the SI property holds for the majority of mainstream applications. On the other hand, should an application not meet the SI property in the other phases, the possible degradation of the guarantees on bandwidth distribution and request completion times would have a negligible impact on the overall application performance.

3.1 Sector-Domain Properties

To show BFQ sector guarantees, we refer to a sector variant of the Bit-Worst-case Fair Index (Bit-WFI), originally defined in packet systems [14]. This index, which we denote as Sector-WFI, allows us to predict the minimum amount of service guaranteed by a system to an application over any time interval during which the application is continuously quasi-backlogged (Section 2). The following theorem holds:

Theorem 1. *For any time interval $[t_1, t_2]$ during which the ith application is continuously quasi-backlogged, BFQ guarantees that*

$$\phi_i \cdot W(t_1,\ t_2) - W_i(t_1,\ t_2) \leq B_{max} + B_{i,max} + L_{max}. \quad (1)$$

The right-hand side in (1) is the Sector-WFI of BFQ. Note that, if an application enjoys the SI property, then the time intervals during which it needs to access the disk safely coincide with the time intervals during which it is quasi-backlogged. Given the strong similarities between WF^{2}Q+ and BFQ, the proof of Theorem 1 is basically an extension of the proof of the Bit-WFI of WF^{2}Q+ [14]. Whereas the full proof is reported in [18], an intuitive justification of each component of the bound follows.

The component B_{max} measures the deviation from the ideal service due to not respecting the batch completion order of the ideal system. More precisely, if the ith application has a lower virtual finish time than the active one, but becomes backlogged *too late*, it may unjustly wait for the service of at most B_{max} sectors before accessing the disk.

The second component, $B_{i,max}$, stems from the fact that if there is no constraint on the request arrival pattern, BFQ guarantees the real system to be in advance in serving the ith application for at most $B_{i,max}$ sectors with respect to the minimum guaranteed service at time t_1. The application may pay back for this extra service during $[t_1, t_2]$. However, it is worth noting that this may happen only if a request R_i^j may arrive before the maximum completion time guaranteed in the ideal system to the previous request R_i^{j-1}. Hence, on the opposite end, if this never occurs, i.e., the application never *asks for* more than its reserved service, the component $B_{i,max}$ is not present at all.

The last term follows from the stepwise approximation of $V(t)$. Basically, due to wrong time-stamping, requests may be erroneously treated as if arrived, with respect to the actual arrival time, after a time interval during which the ideal system might have served at most L_{max} sectors.

It is important to note that the rightmost term in (1) does not grow with the time or the (total) amount of service, hence, the long-term bandwidth distribution is *unconditionally* guaranteed. Furthermore, (1) provides a simple relationship between the short-term bandwidth distribution and the value of the parameters that influence the aggregate disk throughput, as detailed in Section 3.4. Finally, it is easy to prove that no scheduler that exclusively serves applications batch by batch may guarantee a lower Sector-WFI than BFQ. Hence, BFQ provides optimal worst-case bandwidth distribution guarantees among this class of schedulers.

3.2 Time-Domain Properties

The following theorem is the starting point for computing the time guarantees of BFQ. Let $t_1 \leq a_i^j$ be a generic time instant such that the ith application is continuously quasi-backlogged during $[t_1, c_i^j]$. To compute a worst-case upper bound to c_i^j, we assume that all the applications, except for the ith one, ideally start to issue asynchronous requests back-to-back from time t_1 (i.e., without waiting for the completion of their outstanding requests). Moreover, to prevent BFQ from increasing budgets, and hence, boosting the throughput more than it would happen in the actual scenario, we assume that the maximum value of the budget of each application, except for the ith application, is set to its average value in the real scenario. Finally, due to the fact that BFQ conceals delayed arrivals for applications enjoying the SI property, if R_i^j is a delayed synchronous request but the application does enjoy the SI property, in the following theorem, a_i^j can be safely assumed to be equal to the time instant at which R_i^j would have arrived if it had not been delayed:

Theorem 2. *Given a request R_i^j, let $t_1 \leq a_i^j$ be a generic time instant such that the ith application is continuously quasi-backlogged during $[t_1, c_i^j]$. Let T_{agg} be the minimum aggregate disk throughput during an interval $[t_1, c_i^j]$ under the above worst-case assumptions. Finally, let L_i^j be the size of R_i^j and $A_i(t_1, a_i^j + T_{FIFO})$ be the sum of the sizes of the requests issued by the ith application during $[t_1, a_i^j + T_{FIFO})$ plus L_i^j. The following inequality holds:*

$$c_i^j - t_1 \leq \frac{Q_i(t_1^-) + A_i\big(t_1, a_i^j + T_{FIFO}\big) + \big(B_i^l - L_i^j\big) + B_{i,max}}{\phi_i T_{agg}} + \frac{B_{max} + B_{i,max} + L_{max}}{T_{agg}}, \quad (2)$$

where $Q_i(t_1^-)$ is the sum of the sizes of the requests of the ith application not yet completed immediately before time t_1, and B_i^l is (the size of) the budget assigned to the ith application to serve the batch that R_i^j belongs to.

As before, the proof of this theorem, which can be found in [18], is just an extension of the proof of the Time-WFI of WF^{2}Q+. We discuss here the terms in (2), and in Section 3.3 how to use (2) to perform admission control and to provide actual time guarantees.

The right-hand side of (2) can be rewritten as $\frac{1}{\phi_i T_{agg}} \cdot (Q_i(t_1^-) + A_i(t_1, a_i^j + T_{FIFO})) + d_i^j$. It is easy to see that the first component represents the worst-case completion time of R_i^j in an ideal system guaranteeing no lagging behind the reserved service over any time interval. In contrast, d_i^j represents the *worst-case delay* with respect to the ideal worst-case completion time. With regard to the first component, note that, if the ith application issues only synchronous requests, the latter is always served in FIFO order. This is equivalent to assuming $T_{FIFO} = 0$.

The first component of the worst-case delay, $B_i^l - L_i^j$, stems from the fact that the batch that R_i^j belongs to is not time-stamped (and scheduled) as a function of L_i^j, but as a function of B_i^l. Hence, in the worst case, the service of R_i^j may be delayed proportionally to the difference $B_i^l - L_i^j$. Finally, the B_{max}, $B_{i,max}$ (which appears twice in d_i^j), and L_{max} terms can be explained using the same arguments as for the same terms in (1).

3.3 Admission Control and Time Guarantees

We now discuss how to use (2) for performing admission control and providing actual bandwidth and completion time guarantees. First of all, the aggregate throughput T_{agg} must be known at some extent. The tricky aspect is that T_{agg} is in its turn a function of the many user configurable parameters $B_{i,max}$, B_{max}, T_{FIFO}, T_{max}, and T_{wait}. However, as shown in detail in Section 3.4, basing upon (2), the desired trade-off between completion times and T_{agg} can be achieved by iteratively tuning the values either of each of these parameters or of just the throughput boosting-level parameter. The accuracy of the computed guarantees then depends on how accurately T_{agg} itself is known (worst case or average value, variance, confidence interval, ...). In this respect, recall that referring to the expected throughput/service time, albeit unavoidably affected by approximations, is the only option to provide practical time guarantees with any disk scheduling algorithm.

Once known the worst-case throughput, the requirements of an application requesting a long-term throughput T_i and no other type of guarantees can be fulfilled by assigning to it a weight $\phi_i = \frac{T_i}{T_{agg}}$. The application is then admitted only if the resulting sum of the weights $\sum_{q=1}^{N} \phi_q$ is still no higher than one.

In contrast, to provide guarantees on single request completion times to an application, the request arrival pattern of the application needs to be modeled too. A general request arrival model is the periodic or sporadic pattern: the application issues requests with a size of at most Q_i sectors, and with a period or minimum interarrival time of P_i seconds. This pattern models soft real-time applications, e.g., audio or video streaming ones. Since $T_{agg} \cdot P_i$ sectors are transferred during P_i, it follows that, to meet the throughput requirements of the application, it is enough to set $\phi_i = \frac{Q_i}{T_{agg} \cdot P_i}$. As before, for the application to be admitted, the resulting sum of the weights must still be no higher than one.

With regard to the guarantees on request completion times provided to such a type of applications, it is worth noting that P_i coincides with the maximum time needed by the ideal system to complete the last issued request. Hence, at time a_i^j, R_i^j has been certainly completed in the ideal system, and for what is said in the previous two sections, the $B_{i,max}$ component is absent from (2). In the end, the application must tolerate a worst-case delay (jitter):

$$d_{i,max} \equiv \max_j d_i^j \leq \frac{B_{i,max} - L_{i,min}}{\phi_i T_{agg}} + \frac{B_{max} + L_{max}}{T_{agg}}, \qquad (3)$$

where $L_{i,min} \equiv \min_j L_i^j$. In other words, it is possible to meet the requirements of periodic/sporadic soft real-time applications with relative deadlines equal to $P_i + d_{i,max}$. These requirements do match, e.g., (buffered) video and audio streaming applications.

To show possible values of the bounds (2) and (3) in a real system, and to demonstrate the feasibility of interactive and soft real-time applications with BFQ, a Web, DBMS, and Video-on-Demand service are considered in Section 5.

3.4 Throughput Boosting

Larger budgets increase the probability of serving larger bursts of close or even sequential requests, and hence, of achieving a higher throughput. Besides, a large value of T_{FIFO} may boost the throughput in the presence of asynchronous requests. In this respect, also recall that most mainstream applications issue only synchronous requests. Hence, in a system serving this kind of applications, T_{FIFO} has no impact either on the throughput or on the time guarantees (Section 3.2). In contrast, T_{wait} may be just set to the most effective value for the target disk device, equal to the device-dependent average cost of seek and rotational latencies, usually between 4 and 8 ms (and set by default to 4 ms in the current release of BFQ).

BFQ exports a last low-level configuration parameter related to disk throughput, namely the system-wide maximum *time budget* T_{max} (possibly automatically computed, see Section 5.1). Once got access to the disk, each active application must consume all of its time budget or backlog within no more than T_{max} time units; otherwise, it is unconditionally (over)charged for a $B_{i,max}$ service and the next active application is selected. This additional mechanism prevents applications performing random IO from substantially decreasing the disk throughput. Hence, it guarantees practical bandwidths and delays, which are inversely proportional to the aggregate throughput (Section 3.2), to applications performing mostly sequential IO. In contrast, applications performing random IO virtually receive no service guarantees.

According to (1) and (2), in addition to influencing the disk throughput, all these parameters directly or indirectly influence also guarantees. Hence, to set the desired trade-off between guarantee granularity and throughput boosting, the values of these parameters must be (iteratively) tuned by (iteratively) measuring the resulting throughput. BFQ also provides a simplified interface, which allows a user not interested in full control over all the parameters to avoid the resulting tuning complexity. This interface has been used in the experiments reported in Section 5, and is described in detail in Section 5.1.

Whatever interface is used, to evaluate both the (worst-case) throughput and the (worst-case) guarantees as a function of the parameters, and to tune the latter, it is necessary to measure the aggregate throughput against some (worst-case) benchmark request pattern. Such a pattern may be defined as a function of the expected one. For example, the following conservative worst-case pattern may be used to evaluate the expected minimum aggregate throughput for simultaneous sequential reads. After placing two files with size S_{min}, equal to the (portions of the) files that will be interested by sequential accesses, at the maximum possible distance in $[P_{first}, P_{last}]$, where $P_{last} - P_{first}$ is the maximum span of the positions of the requests issued by the applications, a simultaneous sequential read of the two files may be performed. As confirmed also by our experiments, the resulting aggregate throughput provides a lower bound to the aggregate throughput for parallel file reads as well as Web server workloads. For random workloads, a request arrival pattern with the same locality as the expected one can be used to estimate the expected aggregate throughput. An example of the aggregate throughput achieved with a random workload is reported in Section 5.6.

4 Related Work

Existing algorithms for providing a predictable disk service can be broadly divided into three groups: 1) *Real-time* disk schedulers [20], 2) *Proportional share* or *bandwidth reservation* time-stamp-based disk schedulers (also known as *fair-queuing* schedulers), and 3) *Proportional share* round-robin disk schedulers. In addition, examples of frameworks for providing QoS guarantees are Cello [21], APEX [22], PRISM [23], and Argon [24]. It is worth mentioning also real-time operating systems such as the Dresden Real-Time Operating System (DROPS) [5] and RT-Mach [6], and Real-Time Database Systems (RTDBS), which are architectures for performing database operations with real-time constraints [25].

There is no relation between any of the scheduling problems highlighted in this paper, namely loss of throughput and/or guarantees due to deceptive idleness and/or delayed arrivals, and any characteristic of the above-mentioned frameworks for QoS provisioning and RTDBSes (apart from which underlying scheduling algorithm(s) they rely on). Accordingly, after the following note about disk internal queuing, in the next sections, we focus only on each of the above-listed classes of scheduling algorithms.

Disk internal queuing can be used only if disk idling is disabled, which ultimately causes loss of guarantees with any scheduler/framework for applications issuing synchronous requests. Consider, e.g., a proportional share scheduler, and suppose that the ith application has an arbitrarily high weight. After each request R_i^j of the application is completed, the disk starts serving its next internally queued request, or immediately asks for a new request, without waiting for R_i^{j+1} to arrive. Hence, another application is served, independently of whether R_i^{j+1} would then happen to be the next request to serve. In the end, the guarantees assigned to the application may be easily violated.

4.1 Real-Time Schedulers

Real-time schedulers [4], [20] are time-stamp-based schedulers that associate a deadline to each request. They usually start from an Earliest Deadline First (EDF) [26] schedule, and reorder requests to reduce seek and rotational latency without violating deadlines.

Each time the next request to serve must be picked, Rotational-Position-Aware disk scheduling based on a Dynamic Active Subset (SATF-DAS) [5] iteratively constructs and serves a subset of the outstanding requests, called Dynamic Active Subset (DAS) and such that any throughput boosting algorithm can be used to order the requests in the DAS without violating service guarantees. Defined as *slack* of a request, the difference between the deadline of the request and the earliest time by which the request can be served, Just-In-Time Slack Stealing (JIT) [6] is based on serving requests closer to the disk head instead of requests with lower deadlines but with large enough positive slack. SCAN-EDF [4] serves requests in EDF order, but if several requests have the same deadline, they are scheduled using a seek optimization algorithm (e.g., SCAN or C-LOOK). Finally, other proposals are Priority SCAN (PSCAN), Earliest Deadline SCAN, and Feasible Deadline SCAN (FD-SCAN) [25], which are quite similar in principle to the above-described ones.

With regard to throughput boosting, in both SATF-DAS and JIT, there is no control either on the size of the DAS or on the amount of available slack. In contrast, the authors of SCAN-EDF propose enlarging the request size and extending relative deadlines beyond the period to effectively trade response time and buffer requirements for throughput boosting.

Problems arise if applications issuing synchronous requests are scheduled with real-time bandwidth servers built on top of the above real-time schedulers. First, as all of these schedulers are work-conserving, throughput is likely to be very low. However, this problem can be easily solved by extending these schedulers, where possible, to perform disk idling as in BFQ, CFQ, or AS. The second problem follows from the fact that deadlines may be missed, mainly because of the non-preemptability of the service of a request. The consequent delayed arrivals, plus the fact that in bandwidth servers the absolute deadlines of the requests are usually computed by summing their relative deadlines to their arrival times, may cause delayed requests to be unjustly assigned higher deadlines. As discussed Section 1, this may cause the desired bandwidth distribution to be violated. Both problems clearly manifest themselves in our experimental results (Section 5).

4.2 Proportional Share Time-Stamp-Based Schedulers

YFQ [8] dispatches requests to the disk device in *batches*. In particular, before the next batch is served, all the requests in the current batch are dispatched. The requests to insert in each batch are chosen using WFQ [14] and may be ordered with the desired throughput boosting algorithm within the batch. The fact that the limited room in a batch is, in general, filled with requests issued by all the backlogged applications reduces the probability of inserting a high number of requests of the same application. This may reduce the number of close or sequential accesses with mainstream applications performing sequential IO. Moreover, guarantees may be violated for an application issuing synchronous requests: even if the application has a (much) higher weight than the others, no more than one request of the application will be served for each batch.

SFQ(D) and FSFQ(D) [12] allow a configurable number of outstanding requests D to be dispatched, where, each time one of the outstanding requests completes, the next one is immediately dispatched. SFQ(D) selects requests according to a variant of SFQ that does not suffer from loss of fairness in the presence of applications not consuming their fair share of the disk throughput. FSFQ(D) is a further refinement of SFQ(D) that tries to compensate the possible loss of service of an application due to the late arrival of its requests. Similar to SFQ(D) and FSFQ(D), Hybrid [7] allows a configurable amount of outstanding requests to be chosen by an internal WF2Q+ scheduler, and to be reordered by the desired throughput boosting algorithm. To avoid starvation, Hybrid periodically flushes all the outstanding requests. As in BFQ, the internal scheduler works in the service domain. Finally, pClock [9] is based on a more general scheme: through arrival curves, application requirements are expressed in terms of throughput, latency, and maximum burst size. Applications following their arrival curve are proved to never miss their deadlines.

Different from BFQ and Hybrid, YFQ, SFQ(D), FSFQ(D), and pClock directly target proportional time allocation instead of sector allocation, and the accuracy of their disk throughput distribution depends on the accuracy in estimating request service times. The main problem is, however, that, as real-time schedulers, all the schedulers mentioned in this section time-stamp requests as a function of their actual arrival times. Hence, independently of whether disk idling is performed, they may fail to distribute the bandwidth as desired in the presence of synchronous requests. Our experimental results confirm this problem.

4.3 Round-Robin Schedulers

CFQ is a proportional share disk scheduler that grants disk access to each application for a fixed time slice. Slices are scheduled according to a round-robin policy. This time-based allocation, equal to the one adopted in the Argon framework, has the advantage of implicitly charging each application for the seek and rotational latencies it incurs. Unfortunately, this scheme may suffer from unfairness problems also toward applications making the best possible use of the disk bandwidth. Even if the same time slice is assigned to two applications, they may get a different throughput each, as a function of the positions on the disk of their requests. BFQ owes to CFQ the idea of exclusively serving each application for a while, but provides strong guarantees on bandwidth distribution because the assigned budgets are measured in number of sectors.

Strong service distribution guarantees on a per-request basis are provided also by the adaptive Deficit Round-Robin proposed in [10], as it measures the amount of service received by any application in terms of number of requests served. To achieve the desired trade-off between fairness and I/O efficiency, the scheduler is also configurable in terms of maximum number D of outstanding requests and maximum per-application number G_i of requests dispatched in each round. Unfortunately, as any round-robin scheduler, both this scheduler and CFQ are characterized by an $O(N)$ worst-case jitter in request completion time, where N is the number of competing applications. In contrast, due to the accurate service distribution of the internal B-WF^{2}Q+ scheduler, BFQ exhibits $O(1)$ jitter according to (2) with respect to the number of applications. A quantitative evaluation of the consequences of this different short-term guarantees and of the above-mentioned unfairness of CFQ is reported in Section 5.

5 Experimental Results

In this section, we report the results of our single-disk experiments with BFQ, SCAN-EDF, YFQ, CFQ, C-LOOK, and AS, on a system running the Linux 2.6.21 kernel. We first provide implementation and configuration details in the next section. Then we describe the experimental setup and report the results of each set of experiments in the successive ones.

5.1 Scheduler Implementation and Configuration

If the simplified interface of BFQ is used, T_{FIFO} is set to the default value used by the other schedulers in the system (typically 100 ms), T_{max} is dynamically set/updated to $1.\overline{3}$ times the average time needed to consume B_{max} sectors, $\forall i B_{i,max}$ is set to B_{max}, and either a *throughput boosting level* ranging from 0 to 1, or just B_{max}, is exported as the only configuration parameter. In the first case, the back end of the interface will take care of setting B_{max} accordingly, from the minimum possible request size to the number of sectors served in 200 ms. The latter value is automatically computed/updated [16] and guarantees a high throughput, as shown in Section 5.3. We used the simplified interface and set only the maximum budget B_{max} in our experiments.

As no code of SCAN-EDF and YFQ was available for the Linux kernel, we implemented a slightly extended version of each of them in the 2.6.21 Linux kernel [16]. In our implementation of SCAN-EDF, each application is associated with a dynamically configurable relative deadline, equal, e.g., to the application's period. This relative deadline is assigned to each request issued by the application. The resulting algorithm can be seen as a simple real-time bandwidth server. Suppose that the ith application issues requests of the same size L_i back-to-back, and that a relative deadline equal to P_i is assigned to all of its requests. The computed absolute deadlines of the requests will be the same as if the application was periodic with period P_i. Hence, in a full-loaded system, the application should be guaranteed a fraction of the bandwidth equal to L_i/P_i. Finally, to deal with the deceptive idleness, after the completion of the last request of an application, both implementations keep the disk idle until either a new request of the just served application arrives, or a configurable T_{wait} time interval elapses.

To trade response time for throughput boosting, the authors of SCAN-EDF suggest both to change the request size and the coarsen deadline granularity. Unfortunately, the first strategy cannot be used in the Linux kernel, as the request size is not controlled by the disk schedulers. In contrast, our implementation of SCAN-EDF allows a system-wide *granularity* parameter Δ to be set. Given a request with absolute deadline d, and the smallest n such that $n \cdot \Delta \geq d$, the request is scheduled as if its deadline was $n \cdot \Delta$. With regard to YFQ, the batch size, measured in number of requests, is configured through the BT_{size} parameter. Batch overlapping, one of the enhancements proposed by the authors of YFQ to increase disk throughput, is performed as well (see the code [16]). All the requests with the same deadline and all the requests in a batch are served in C-LOOK order.

5.2 Experiments

The experiments were aimed at measuring the aggregate throughput, long-term bandwidth distribution, and (short-term) per-request completion time guaranteed by the six schedulers with the following applications/workloads: simultaneous sequential reads, Web server (emulated), DBMS (emulated), and mixed video-streaming/file reads. All these applications only issue synchronous requests on a Linux system. In addition, as discussed at the end of Section 5.3, the same results in terms of aggregate throughput would be achieved with all these applications, apart from DBMS, in case they would issue asynchronous requests. Hence, only in the DBMS case, we also showed the different performances of the schedulers with an asynchronous workload. Due to the space limitations, only a synthesis of the results is reported here. The complete results and all the programs used to generate them can be found in [16].

We ran the experiments on a PC equipped with a 1 GHz AMD Athlon processor, 768 MB RAM, and a 30 GB IBM-DTLA-307030 ATA IDE hard drive (roughly 36 MB/sec peak bandwidth in the outer zones, ~ 35 percent lower throughput in the inner zones), accessed in UDMA mode. Using this low-performance disk device helped us guarantee that the disk was the only bottleneck. Sectors were 512 bytes long (*ext2* file-system). The disk was partitioned into 30 consecutive slices of equal size, the first slice covering the outer part of the disk and the last one covering the inner part. All the programs were run from an auxiliary disk. For each type of experiment and set of values of the parameters, the same experiment was repeated 20 times (the buffer cache was flushed before each experiment). The minimum, maximum, and mean value, together with its associated 95 percent confidence interval, were computed for each output quantity. In what follows any mean value v is reported in the form $v \pm s$, where s is the semiwidth of the 95 percent confidence interval for v.

5.3 Aggregate Throughput

The first set of experiments was aimed at estimating the worst-case aggregate throughput guaranteed by each scheduler in case of simultaneous sequential reads. As can be seen in the next sections, according to our experiments, these results hold also for a Web server workload. Under BFQ and YFQ, all applications were assigned the same weight, whereas they were assigned the same priority under CFQ (which allows applications to be assigned different priorities). Under SCAN-EDF, all the requests were assigned the same deadline, equal to 20 ms. The whole set of different experiments was given by the combinations of the following five values: scheduler in {BFQ, SCAN-EDF, YFQ, CFQ, C-LOOK, AS}; cardinality of the set of distinct files to read in {2, 3, 4, 5} (for each set, the files were placed in slices at the maximum possible distance from each other, with each file in a distinct slice); value of the scheduler *configuration parameter*: maximum budget B_{max} in {512, 1,024, 2,048, 4,096, 8,192, 16,384} sectors for BFQ, batch size BT_{size} in {4, 8, 16} requests for YFQ, deadline granularity Δ in {20, 40, 80, 160, 320} ms for SCAN-EDF, and time slice T_{slice} equal to 100 ms (the default value) for CFQ; T_{wait} in $\{0, 4\}$ ms for SCAN-EDF and YFQ, and $T_{wait} = 4$ ms for BFQ (CFQ automatically sets/changes T_{wait}); size of every file in {128, 256, 512, 1,024} MB. Of course, T_{wait} is implicitly 0 ms in C-LOOK, whereas it was set to 4 ms in AS with any additional fairness policy deactivated. The minimum file size was (experimentally) chosen so as to let the results be due only to the disk schedulers, without significant distortions due to unrelated short-term factors such as CPU scheduling.

TABLE 2
Aggregate Throughput for Two Simultaneous Reads

Scheduler	Mean Agg. Thr [MB/s]	Value of B_{max} T_{slice}, Δ, BT_{size}
BFQ	22.46 ± 0.81	16384 sect
SCAN-EDF		
$T_{wait} = 0$ ms	21.18 ± 0.47	640 ms
$T_{wait} = 4$ ms	23.39 ± 0.51	
YFQ		
$T_{wait} = 0$ ms	10.64 ± 0.25	16 reqs
$T_{wait} = 4$ ms	10.80 ± 0.20	
CFQ	16.91 ± 1.30	100 ms
C-LOOK	20.59 ± 0.76	
AS	32.97 ± 1.89	

With any scheduler, the lowest throughputs were achieved in case of two, 128 MB long, files, most certainly because in this case, the disk head covers the longest distance and (spends more time moving) between the files (the influence of the length of the files was in the order of a few tenths of an MB/s). For this scenario, Table 2 reports both the maximum value of the mean aggregate throughput achieved by the six schedulers, and the value of the configuration parameter for which this value was achieved.

Whereas the highest throughput is achieved by AS, the bad performance of C-LOOK is due to the fact that it frequently switches from one file to the other, for it does not perform disk idling. It is easy to see that, for $B_{max} =$ 16,384 sectors, a maximum length budget is served in about 200 ms under BFQ, which confirms that a high throughput is achieved if a throughput boosting level of one is set with the simplified interface. Moreover, the higher throughput achieved by BFQ and SCAN-EDF with respect to CFQ results from the higher number of (sequential) sectors of a file that can be read before switching to the other file. More precisely, considering the disk peak rate, it is easy to see that less than $16,384$ sectors can be read in 100 ms (which is the value of T_{slice} for CFQ), whereas more than $16,384$ sectors can be read in 640 ms (which is the value of Δ for SCAN-EDF). Notably, T_{wait} does not influence much the aggregate throughput with SCAN-EDF. In fact, as the system performs read-ahead for sequential accesses, it tends to asynchronously issue the next request before the completion of the current one (this also helps C-LOOK). Finally, the poor performance of YFQ and the fact that it is independent of both T_{wait} and the batch size (not shown) confirm the arguments in Section 4.2.

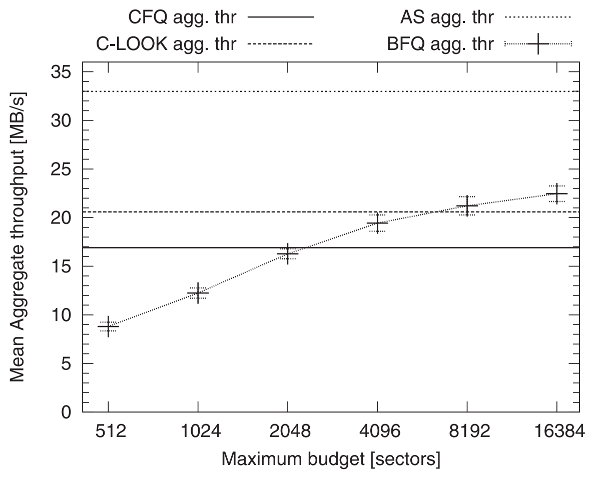

Fig. 5. Mean aggregate throughput, and associated 95 percent confidence interval, achieved by BFQ (as a function of B_{max}), CFQ, C-LOOK, and AS in case of simultaneous reads of two, 128 MB long, files.

According to the observations in Section 3.4, to evaluate the possible trade-offs between guarantee granularity and aggregate throughput with BFQ, it is necessary to know how the throughput varies as a function of B_{max} in the worst-case scenario. This piece of information is shown in Fig. 5 in case the simultaneous sequential reads of two, 128 MB long, files are used as worst-case pattern. The aggregate throughputs of CFQ, C-LOOK, and AS are reported as a reference too. For $B_{max} = 4{,}096$ sectors, BFQ guarantees a higher throughput than CFQ. Finally, it can be seen that the aggregate throughput with AS is close to the disk peak rate, which means that exclusively serving each application for a certain amount of time—as AS, CFQ, and BFQ do—leads, in practice, to the highest possible throughput if applications issue sequential requests. This also implies that SCAN-EDF and YFQ certainly would not achieve a higher throughput if these applications issued asynchronous requests.

5.4 Simultaneous Sequential Reads

The second feature we evaluated is the accuracy of the schedulers in providing the desired bandwidth fraction to applications performing sequential reads. For brevity, for BFQ, we only report the results for $B_{max} = 4{,}096$ sectors. Similarly, for SCAN-EDF, we consider only $\Delta = 80$ ms because for this value, SCAN-EDF achieves an aggregate throughput close to BFQ. Finally, in our experiments, the batch size had no influence on the guarantees provided by YFQ. We report the results for $BT_{size} = 4$ requests.

We first considered the case where all the applications are allocated the same fraction of the disk bandwidth. In particular, during the same experiments used to evaluate the aggregate throughput, we measured the throughput of each (file read) application. For each scheduler, the highest deviation from the ideal distribution occurred for 128 MB long files. Moreover, we observed that the 95 percent confidence interval for the mean throughput of the ith application in case of three and four files was always smaller (or greater) than that obtained in case of five (or two) files. This *inclusion property* held also in case of asymmetric allocation (see below). Hence, for brevity, we report only the results for two and five, 128 MB long, files. Finally, we consider only $T_{wait} = 4$ ms for SCAN-EDF and YFQ in Table 3.

TABLE 3
Simultaneous Sequential Reads

Throughput (2 files)	9.95 ±0.43				9.81 ±0.47
Throughput (5 files)	4.29 ±0.10	4.30 ±0.09	4.30 ±0.07	4.29 ±0.10	4.31 ±0.09
BFQ (B_{max} = 4096 sectors)					
Throughput (2 files)	10.72 ±0.22				9.62 ±0.19
Throughput (5 files)	2.17 ±0.02	2.17 ±0.02	2.17 ±0.02	2.17 ±0.02	2.19 ±0.03
SCAN-EDF (Δ = 80 ms, $T_{wait} = 4$ ms)					
Throughput (2 files)	5.44 ±0.10				5.44 ±0.10
Throughput (5 files)	1.39 ±0.02	1.39 ±0.02	1.39 ±0.02	1.39 ±0.02	1.39 ±0.03
YFQ (BT_{size} = 4 requests, $T_{wait} = 4ms$)					
Throughput (2 files)	11.92 ±0.44				8.61 ±0.67
Throughput (5 files)	5.24 ±0.14	4.91 ±0.15	4.66 ±0.11	4.37 ±0.10	4.01 ±0.17
CFQ (T_{slice} = $100ms$)					
Throughput (2 files)	11.52 ±1.78				10.57 ±0.55
Throughput (5 files)	5.56 ±0.67	5.14 ±0.41	5.05 ±0.50	4.95 ±0.35	4.83 ±0.09
C-LOOK					
Throughput (2 files)	32.41 ±13.39				17.78 ±1.10
Throughput (5 files)	32.83 ±11.95	31.72 ±1.37	29.80 ±0.50	12.56 ±10.46	18.29 ±0.43
AS					

BFQ, YFQ, and C-LOOK exhibit the most accurate bandwidth distribution (C-LOOK basically performs a round-robin among the batch of requests issued by the read-ahead mechanism). Unfortunately, as previously seen, YFQ has a low throughput. SCAN-EDF is less accurate for two files, but it provides a higher throughput than YFQ. Consistently with the arguments in Section 4.2, CFQ fails to fairly distribute the throughput because of the varying sector transfer speed. Finally, as expected, AS has the most asymmetric bandwidth distribution. Moreover, the high width of the confidence interval for AS is a consequence of the fact that sometimes the waiting timer may expire before the next synchronous request arrives. In that case, also AS switches to the service of another application.

It is important to observe that the accurate throughput distribution of YFQ and SCAN-EDF is mostly related to the symmetry of the bandwidth allocation. To measure the accuracy of the schedulers in distributing the disk bandwidth in case of asymmetric allocations, under BFQ and YFQ, we assigned different weights to the applications. We run two sets of experiments, using, respectively, the weights 1 and 2, and the weights 1, 2, and 10 (there is no constraint on the values of the weights in our implementations of BFQ and YFQ). Moreover, assuming that the jth application is the one

TABLE 4
BFQ with Asymmetric Weights

Weight	2	2	2	1	1
Throughput (2 files)	14.62 $\pm$0.60				7.31 $\pm$0.32
Throughput (5 files)	5.60 $\pm$0.08	5.60 $\pm$0.08	5.60 $\pm$0.08	2.81 $\pm$0.04	2.81 $\pm$0.04
Weight	10	2	2	1	1
Throughput (2 files)	26.81 $\pm$0.60				2.77 $\pm$0.16
Throughput (5 files)	16.06 $\pm$0.28	3.25 $\pm$0.07	3.25 $\pm$0.07	1.64 $\pm$0.04	1.65 $\pm$0.04

TABLE 5
Web Server (Random/Sequential Reads)

Scheduler	**Compl. time small files [sec]**	**Bandw large files** [sec]	**Mean aggr. throughput** [MB/s]
BFQ	1.74 ± 0.11	2.26 ± 2.34	17.13 ± 0.65
SCAN-EDF	7.68 ± 0.20	3.29 ± 3.10	8.71 ± 0.09
YFQ	0.40 ± 0.01	1.36 ± 1.71	6.87 ± 0.09
CFQ	3.86 ± 0.17	2.97 ± 2.47	18.20 ± 0.79
C-LOOK	7.78 ± 0.27	9.65 ± 3.93	19.47 ± 0.66
AS	7.69 ± 0.27	16.61 ± 4.00	20.60 ± 1.22

with maximum weight, and denoted by S_j the size of the file read by the jth application, the file read by the ith application had a length $\frac{\phi_i}{\phi_j} S_j$. To try to allocate the same bandwidth as with BFQ and YFQ, under SCAN-EDF, we assigned to each request of an application a relative deadline inversely proportional to the weight assigned to the application with the other two schedulers. Finally, this type of experiments was not run for CFQ, C-LOOK, and AS, which do not provide differentiated bandwidth allocations.

To show the worst case, yet not distorted by external factors, performance of the schedulers, for the scenarios where the maximum weight was, respectively, 2 and 10, we report the results of only the experiments where the maximum file sizes were, respectively, 256 MB and 1 GB. Finally, since the above-defined inclusion property holds also in case of asymmetric allocation, for brevity, we report the results with BFQ only for the two and five files scenarios in Table 4.

With regard to SCAN-EDF and YFQ, in accordance with what is said in Sections 4.1 and 4.2, both failed to guarantee the desired bandwidth distribution in all the experiments. For example, in case of two applications with weights 10 and 1, the throughputs were $\{26.58 \pm 0.50, 6.92 \pm 0.20\}$ MB/s for SCAN-EDF, and $\{22.70 \pm 0.12, 5.41 \pm 0.12\}$ MB/s for YFQ (the skewness is even attenuated by the fact that the application with a higher weight has to read a longer file, and hence, it gets exclusive access to the disk after the other one finished).

5.5 Web Server

In this set of experiments, we estimated the per-request completion time guaranteed by the schedulers against the following Web server-like workload: 100 processes (all with the same weight/priority) continuously read files one after the other. Each of the files to read may have been, with probability 0.9, a small 16 KB (html) file at a random position in the first half of the disk, or with probability 0.1, a large file with random size in $[1, 30]$ MB at a random position in the second half of the disk. Every 10 files, each process appended a random amount of bytes, from 1 to 16 KB, to a common log file. Such a scenario allows the performance of the schedulers to be measured for a mainstream application, and in general, in the presence of a mix of both sequential (large files) and a random (small files) requests. Especially, for each run, lasting for about one hour, we measured the completion time of each small file (latency), the (average) bandwidth at which large files were read, and the average aggregate throughput (measured every second). In this respect, small and large file reads were performed in separated parts of the disk to generate an asymmetric workload, which is more prone to higher latencies and/or lower bandwidths.

As can be seen from Table 5—excluding for a moment SCAN-EDF and YFQ—BFQ, CFQ, and AS stand, respectively, at the beginning, (about) the middle, and the end of the low latency versus high aggregate throughput scale. Also C-LOOK achieves similar performance as AS because a high number of competing requests scattered over all the entire disk are present at all times. In contrast, YFQ has very low latencies and throughput because, as the probability of a small file read is nine times higher than the one of a large file read, each batch is likely to contain a high percentage of requests pertaining to small files. Finally, although achieving a lower aggregate throughput than BFQ, SCAN-EDF guarantees higher bandwidths to the large files, by sacrificing the latency of the small ones. It is worth noting that the observed mean latency of BFQ is 2.22 times lower than CFQ and at least 4.41 times lower than all the other schedulers. On the contrary, with any scheduler, the high width of the confidence interval for the mean bandwidth of the large files is a consequence of the quite random nature of the workload.

To check whether the guarantee (2) complies with the observed latencies for small files, we can set t_1 to the arrival time of the first request of a generic small file, and assume that R_i^j is the last request for that file, which implies $Q_i(t_1^-) + A(t_1, a_i^j) = 16kB$ (as explained in Section 3.2, T_{FIFO} can be neglected with synchronous requests). Moreover, $\phi_i = \frac{1}{100}$, $L_{max} = 256$ sectors in the Linux kernel (by default), and in the experiments we found that if the ith application is one of the processes reading small files, then $L_{i,min} \equiv \min_j L_i^j = 16$ sectors and a budget oscillating from 8 to 256 sectors is assigned to the application. Hence, setting $B_{i,max} = 256$ sectors and using the mean throughput $T_{agg} = 17.13$, the resulting guaranteed latency is

$$\frac{100 * (16 + (128 - 8) + 128) + (2,048 + 2 * 128)}{17.13 * 1,024} = 1.64 \text{ seconds.}$$

This value is approximately six percent lower than the observed mean latency, which is most certainly due to the fact that when small files are read, the throughput falls down to a lower value than the mean used in the formula.

TABLE 6
DBMS (Random Reads)

Scheduler	Num. of outstand. requests	Completion time [sec]	Mean aggr. throughput [MB/s]
BFQ	1	0.92 ± 0.12	0.44 ± 0.00
	10	7.93 ± 0.29	0.49 ± 0.00
SCAN-EDF	1	0.93 ± 0.00	0.44 ± 0.00
	10	6.49 ± 0.85	0.60 ± 0.00
YFQ	1	0.69 ± 0.01	0.59 ± 0.00
	10	6.59 ± 0.06	0.59 ± 0.00
CFQ	1	0.91 ± 0.09	0.45 ± 0.00
	10	8.01 ± 0.30	0.49 ± 0.00
C-LOOK/	1	0.67 ± 0.01	0.62 ± 0.00
AS	10	5.52 ± 0.12	0.71 ± 0.00

TABLE 7
Video Streaming Disturbed by Sequential Reads

Scheduler	Param.	Mean Num. of Movies	Mean Agg. Thr. [MB/s]
BFQ	4096 sect	24.00 ± 0.00	7.56 ± 0.87
	8192 sect	23.95 ± 0.42	8.15 ± 1.08
	16384 sect	18.70 ± 9.45	12.78 ± 5.64
SCAN-EDF	20 ms	12.00 ± 0.00	8.93 ± 0.22
YFQ	4 reqs	19.00 ± 0.00	5.85 ± 0.55
CFQ	20 ms	14.35 ± 1.40	12.59 ± 2.12
C-LOOK		1.8 ± 1.16	22.66 ± 0.96
AS		1.1 ± 1.04	28.39 ± 5.36

5.6 DBMS

This set of experiments was aimed at estimating the request completion time and the aggregate throughput achieved by the schedulers against a DBMS-like workload: 100 processes (all with the same weight/priority) concurrently issue direct (noncacheable) 4 KB read requests at random positions in a common 5 GB file. We have used this setup to evaluate also the performance of the schedulers with asynchronous requests, and to this purpose, we have run 10 variants of the experiment, with each variant characterized by a different number of per-process outstanding requests, ranging from one (synchronous requests) to 10. According to our results, for each scheduler, both the request completion time and the aggregate throughput monotonically grew with the number of outstanding requests. Hence, for brevity, we report in Table 6 the mean request completion time and aggregate throughput only for one and 10 outstanding requests.

With random requests, AS does not perform disk idling at all, hence, it achieves the same results as pure C-LOOK. Especially, due to their global position-based request ordering, C-LOOK and AS achieve the lowest request completion time, and hence, the highest disk throughput. Being its service scheme closer to C-LOOK/AS than the one of BFQ, SCAN-EDF, and CFQ, YFQ outperforms the latter in case of synchronous requests. It has instead the same performance as SCAN-EDF with 10 outstanding requests. Finally, because of their slice by slice/budget by budget service scheme, both CFQ and BFQ exhibit an approximately 1.4 times higher request completion time/lower throughput than C-LOOK/AS. It is, however, worth noting that only approximately two percent of the disk rate is achieved by the latter, which is typically unbearable in a realistic DBMS. This result complies with the fact that, if the disk requests enjoy some locality, caches are usually tuned so as to reduce disk access and achieve feasible response times (of course, multiple disks are typically used as well).

5.7 Video Streaming

The last set of experiments was aimed at measuring the ability of the schedulers to support a very time-sensitive application. For this purpose, we set up a VLC streaming server [27], provided with 30 movies to stream to remote clients. Each movie was stored in a distinct disk slice, and the streaming thread of the server that read it was considered as a distinct application by the disk schedulers (all the threads were assigned the same weight/priority). To evaluate the performance of the schedulers, a packet sent by the streaming thread was considered lost if delayed by more than one second with respect to its due transmission time (i.e., we assumed a one-second playback buffer on the client side).

Every 15 seconds, the streaming of a new movie was started. Each experiment ended either if 15 seconds elapsed from the start of the streaming of the last available movie or if the packet loss rate reached the one percent threshold (this value, as well as the one second threshold for the delay, has been experimentally chosen to trade off between achieving a very high video quality and allowing the system to stream a high enough number of simultaneous films to clearly show the different performances of the six schedulers). To mimic the mixed workload of a general purpose storage system and to increase the workload so that the disk was the only bottleneck, during each experiment, we also run five ON/OFF file readers, each reading a portion of random length in $[64, 512]$ MB of a file, and then sleeping for a random time interval in $[1, 200]$ ms before starting to read a new portion of the same file (each file was stored in a different disk slice). Δ, BT_{size}, and T_{slice} for SCAN-EDF, YFQ, and CFQ were set to 20 ms, four requests, and 20 ms, as these are the maximum values for which these schedulers achieve the highest number of simultaneous streams. Table 7 shows the mean number of movies and the mean aggregate throughput achieved by the schedulers during the last five seconds before the end of each experiment.

As can be seen, with $B_{max} = 4{,}096$ sectors, BFQ guarantees a stable and higher number of simultaneous streams than all the other five schedulers. Interestingly, with BFQ, also the aggregate throughput is comparable/higher than SCAN-EDF/YFQ. Most certainly, this is a consequence of the low Δ with SCAN-EDF.

Finally, to check whether the worst-case delay $d_{i,max}$ guaranteed by BFQ to the periodic soft real-time application represented by any of the concurrent VLC streams complies with the observed maximum delay, from Table 7, we can consider that, just before the end of most experiments, in case of $B_{max} = 4{,}096$ sectors, $24 + 5$ applications with equal weights are competing for the disk, and the mean throughput is 7.56 MB/s. As in the Web server experiments, we have that $L_{max} = 256$ sectors, $L_{i,min} \equiv \min_j L_i^j = 16$ sectors, and a budget never higher than 256 sectors is assigned to the generic ith streaming thread of the video server. Hence, according to (3),

$$d_{i,max} \leq \frac{29 * (128 - 8) + (2,048 + 128)}{7.56 * 1,024} = 0.73 \text{ seconds.}$$

This value complies with the above-mentioned 1-second threshold for considering a packet as late, assuming that a reasonable additional worst-case delay of ~ 0.27 second is added by the rest of the system (probably mostly due to the execution of the 29 threads on the CPU).

6 Conclusions

In this paper, we dealt with the problem of providing service guarantees while simultaneously achieving a high disk throughput in the presence of synchronous requests. This type of requests may cause work-conserving scheduler to fail to provide a high throughput, and time-stamp-based schedulers to fail to enforce guarantees.

In this respect, we proposed BFQ, a new disk scheduler that combines disk idling and time stamp back-shifting to achieve a high throughput and preserve guarantees also in the presence of synchronous requests.

References

[1] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," *Proc. 18th ACM Symp. Operating Systems Principles,* Oct. 2001.
[2] G. Lawton, "Powering Down the Computing Infrastructure," *Computer,* vol. 40, no. 2, pp. 16-19, Feb. 2007.
[3] B.L. Worthington, G.R. Ganger, and Y.N. Patt, "Scheduling Algorithms for Modern Disk Drives," *Proc. ACM SIGMETRICS,* pp. 241-251, 1994.
[4] A.L.N. Reddy and J. Wyllie, "Disk Scheduling in a Multimedia I/O System," *Proc. First ACM Int'l Conf. Multimedia (MULTIMEDIA '93),* pp. 225-233, 1993.
[5] L. Reuther and M. Pohlack, "Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS)," *Proc. 24th IEEE Int'l Real-Time Systems Symp. (RTSS '03),* p. 374, 2003.
[6] A. Molano, K. Juvva, and R. Rajkumar, "Real-Time Filesystems. Guaranteeing Timing Constraints for Disk Accesses in RT-Mach," *Proc. 18th IEEE Real-Time Systems Symp.,* pp. 155-165, Dec. 1997.
[7] L. Rizzo and P. Valente, "Hybrid: Achieving Deterministic Fairness and High Throughput in Disk Scheduling," *Proc. CCCT '04,* 2004.
[8] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk Scheduling with Quality of Service Guarantees," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems (ICMCS '99),* vol. 2, p. 400, 1999.
[9] A. Gulati, A. Merchant, and P.J. Varman, "Pclock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems," *SIGMETRICS Performance Evaluation Rev.,* vol. 35, no. 1, pp. 13-24, 2007.
[10] A. Gulati, A. Merchant, M. Uysal, and P.J. Varman, "Efficient and Adaptive Proportional Share I/O Scheduling," technical report, Hewlett-Packard, http://www.hpl.hp.com/techreports/2007/HPL-2007-186.pdf, Nov. 2007.
[11] http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.pdf, 2010.
[12] W. Jin, J.S. Chase, and J. Kaur, "Interposed Proportional Sharing for a Storage Service Utility," *Proc. SIGMETRICS '04/Performance '04,* pp. 37-48, 2004.
[13] http://google-opensource.blogspot.com/2008/08/linux-disk-scheduler-benchmarking.html, 2010.
[14] J.C.R. Bennett and H. Zhang, "Hierarchical Packet Fair Queueing Algorithms," *IEEE/ACM Trans. Networking,* vol. 5, no. 5, pp. 675-689, Oct. 1997.
[15] D. Stephens, J. Bennett, and H. Zhang, "Implementing Scheduling Algorithms in High-Speed Networks," *IEEE J. Selected Areas Comm.,* vol. 17, no. 6, pp. 1145-1158, June 1999.
[16] http://algo.ing.unimo.it/people/paolo/disk_sched, 2008.
[17] D. Stiliadis and A. Varma, "A General Methodology for Designing Efficient Traffic Scheduling and Shaping Algorithms," *Proc. IEEE INFOCOM,* vol. 1, pp. 326-335, 1997.
[18] "Proofs of Theorems 1 and 2," technical report, http://algo.ing.unimo.it/people/paolo/disk_sched/bfq-techreport.pdf.
[19] P. Valente, "Extending WF^{2}Q+ to Support a Dynamic Traffic Mix," *Proc. First Int'l Workshop Advanced Architectures and Algorithms for Internet Delivery and Applications 2005 (AAA-IDEA '05),* pp. 26-33, June 2005.
[20] S. Daigle and J. Strosnider, "Disk Scheduling for Multimedia Data Streams," *Proc. IS&T/SPIE,* pp. 212-293, 1994.
[21] P.J. Shenoy and H.M. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems," *SIGMETRICS Performance Evaluation Rev.,* vol. 26, no. 1, pp. 44-55, 1998.
[22] T.P.K. Lund and V. Goebel, "APEX: Adaptive Disk Scheduling Framework with QoS Support," *Multimedia Systems,* vol. 11, no. 1, pp. 45-59, 2005.
[23] A.L.N. Reddy, J. Wyllie, and K.B.R. Wijayaratne, "Disk Scheduling in a Multimedia I/O System," *ACM Trans. Multimedia Computing, Comm. and Applications,* vol. 1, no. 1, pp. 37-59, 2005.
[24] M. Wachs, M. Abd-El-Malek, E. Thereska, and G.R. Ganger, "Argon: Performance Insulation for Shared Storage Servers," *Proc. Fifth USENIX Conf. File and Storage Technologies,* 2007.
[25] B. Kao and H. Garcia-Molina, "An Overview of Real-Time Database Systems," *Advances in real-time systems,* pp. 463-486 Prentice-Hall, 1995.
[26] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 46-61, 1973.
[27] http://www.videolan.org/vlc/, 2010.

Paolo Valente received the Laurea and PhD degrees in computer systems engineering from the University of Pisa, Italy, in 2000 and 2004, respectively. Since 2006, he is an assistant professor (ricercatore) in the Department of Computer Science at the University of Modena, Italy. His research interests include the design and analysis of CPU, network, and disk scheduling algorithms.

Fabio Checconi received the degree in computer engineering from the University of Pisa in 2005 and is currently working toward the PhD degree in the Real-Time Systems Laboratory, Scuola Superiore Sant'Anna. His research interest includes real-time operating systems.