Proportional Share Scheduling in General Purpose Operating Systems: Theory and Practice

Fabio Checconi

ReTiS Lab.

Scuola Superiore Sant'Anna

A thesis submitted for the degree of

Doctor of Philosophy

Supervisor: Prof. Giuseppe Lipari

22nd of January, 2010

Contents

Introduction

IN this work we propose two novel schedulers based on the proportional share paradigm, both aimed at bringing the benefits of Quality of Service provisioning into general purpose operating systems.

From the point of view of storage systems, mainstream applications such as file copy/transfer, Web, DBMS, or video streaming, typically issue synchronous disk requests. As we will show, this fact may cause work-conserving schedulers to fail both to enforce guarantees and to 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 timestamp-based schedulers, because of the rules they use to tag requests.

Budget Fair Queueing (BFQ), the new disk scheduler we introduce in this work, is an example of how disk idling, combined with proper *back-shifting* of request timestamps, may allow a timestamp-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.

On the other hand, one of the key issues in providing tight bandwidth and delay guarantees in high speed networks is the cost of packet scheduling. Through timestamp rounding and flow grouping several schedulers have been designed that provide near-optimal bandwidth distribution with $O(1)$ time complexity. However, in one of the two lowest complexity proposals in the literature, the cost of each packet enqueue/dequeue is still proportional to the number of groups in which flows are partitioned. In the other proposal, this cost is independent of the number of groups, but a number of operations proportional to the length of the packet being transmitted must be executed during each dequeue operation.

The other scheduler we present, Quick Fair Queueing (QFQ), is a new member of this class of schedulers that provides the same near-optimal guarantees, yet has a constant cost also with respect to the number of groups and the packet length. The peculiarity of QFQ is to partition groups so that enqueue/dequeue can be accomplished in a number of steps independent of the number of groups, without requiring extra operations during packet transmissions.

CHAPTER 1

Proportional Share Scheduling

THIS chapter gives some definitions commonly used in the (packet) scheduling literature that we will reuse in the following chapters, along with some background on proportional share scheduling techniques. We focus on packet scheduling because the theoretical foundations of our work come from there. Even when in Chapter 2 we deal with disk scheduling, we still base our findings on the results shown here.

1.1. System Model and Common Definitions

We consider a system in which N packet flows (defined in any meaningful way) share a common transmission link serving one packet at a time. The link has a time-varying rate, which the system can decide to use, partially or completely, to transmit packets waiting for service. A system is called *work conserving* if the link is used at full capacity whenever there are packets queued. A scheduler sits between the flows and the link: arriving packets are immediately enqueued, and the next packet to serve is chosen and dequeued by the scheduler when the link is ready. The interface of the scheduler to the rest of the system is made of one packet `enqueue()` and one packet `dequeue()` function.

In our model, each flow k is assigned a fixed weight $\phi^k > 0$. Without losing generality, we can assume that $\sum_{k=1}^{N} \phi^k \leq 1$. Proportional share schedulers serve flows according to their *reserved shares*; the reserved share of a flow is proportional to its weight, and corresponds to the minimum fraction of the service provided by the system that should be guaranteed to that flow. A flow is said to be *backlogged* if it owns packets not yet completely transmitted, otherwise we say that the flow is *idle*. We denote as $B(t)$ the set of flows backlogged at time t. Inside the system each flow has a FIFO queue associated with it, holding the flow's own backlog.

We call *head packet* of a flow the packet at the head of the queue, and l^k its length; $l^k = 0$ when a flow is idle. We say that a flow is *receiving service* if one of its packets is being transmitted. Both the amount of service $W^k(t_1, t_2)$ received by a flow and the total amount of service $W(t_1, t_2)$ delivered by the system in the time interval $[t_1, t_2]$ are measured in number of bits transmitted during the interval.

For convenience, the symbols we use more often are listed in Table 1. Most quantities are a function of time, but we omit the time argument (t) when not ambiguous and clear from the context.

1.2. Generalized Processor Sharing

Generalized Processor Sharing [**PG92**] is an ideal scheduling policy that serves all the backlogged flows in parallel, allocating them their exact reserved share. In other words, each backlogged flow k at any time t is guaranteed to receive service at rate

Symbol	Meaning
N	Total number of flows
L	Maximum length of any packet in the system
$B(t)$	The set of backlogged flows at time t
$W(t_1, t_2)$	Total service delivered by the system in $[t_1, t_2]$
k	Flow index
L^k	Maximum length of packets in flow k
ϕ^k	Weight of flow k
l^k	Length of the head packet in flow k, $l^k = 0$ when idle
$Q^k(t)$	Backlog of flow k at time t
$W^k(t_1, t_2)$	Service received by flow k in $[t_1, t_2]$
$V(t)$	System virtual time
S^k, F^k	Virtual start and finish times of flow k, see (2)

TABLE 1. Definitions.

$$\mathrm{d}W^k(t) = \frac{\phi_k}{\sum_{i \in B(t)} \phi_i} \mathrm{d}W(t).$$

While such a scheduling discipline cannot be implemented in practice, it constitutes the term of comparison for the evaluation of real schedulers; to measure how well a scheduler performs, as we will see, the service it provides is compared against the ideal, perfectly fair GPS service. Since GPS scheduling is used as a reference to assess the optimality of a scheduling policy, the members of a wide class of schedulers (see Section 1.4) uses an internal GPS simulation to take their decisions, to stay as close as possible to the ideal schedule.

1.2.1. Fairness Metrics. Packet schedulers are evaluated based on their time (and space) complexity and on their service properties. Several service metrics have been defined in the literature, including *relative fairness* [**Kar06**], and *Bit-* and *Time- Worst-case Fair Index* (B-WFI and T-WFI) [**BZ96**, **BZ97**].

B-WFI^k and T-WFI^k (we will refer to both on them as WFI for brevity) represent the worst-case deviation over any time interval that a flow k may experience, in terms of service and time, with respect to the service it would receive in an ideal system providing perfect weighted bandwidth sharing. They are defined as follows:

$$\begin{aligned} \text{B-WFI}^k &\equiv \max_{[t_1,t_2]} \phi^k W(t_1, t_2) - W^k(t_1, t_2), \\ \text{T-WFI}^k &\equiv \max \left(t_c - t_a - \frac{Q^k(t_a^+)}{\phi^k R} \right), \end{aligned}$$

with t_a being the arrival time of the packet being considered, and t_c being its completion time.

The WFI is an interesting measure as the B-WFI can be also used to predict the minimum amount of service guaranteed to each flow over any time interval, whereas, if the arrival pattern is known, the T-WFI can also be used to compute the maximum delay experienced by each packet. In addition, as proven in [**BZ97**], a low WFI is essential to provide tight fairness and delay guarantees in a hierarchical setting.

1.3. Round Robin Schedulers

Round robin schedulers assign service slots to flows in some sort of round robin fashion. Their simple structure, usually requiring only few cheap list insertions and extractions per each packet enqueue or dequeue allows efficient $O(1)$ implementations to be deployed; at the same time, their simple structure produces large delay bounds and burstiness in the service they provide.

As an example of an early round robin scheduler we present *Deficit Round Robin* (DRR) [**SV95**], which we will use as a term of comparison in Chapter 3. DRR assigns a quantum of service to each flow, in proportion to the flow's weight; the scheduler maintains a deficit counter per each flows, which represents the service the flow is entitled to in the current round of service. Backlogged flows are served as long as they have enough deficit to transmit their next packet; when they cannot transmit no more packets they are postponed to the next round, and a new quantum is added to their deficit counter. Despite being extremely fast, this scheme makes each flow, irrespective of its weight, wait for all the other $N-1$ flows to be served between two consecutive rounds of service.

To prevent this bursty behavior, several enhancements have been proposed. For example, *Smoothed Round Robin* (SRR) [**Chu01**] spreads the quantum allocation over each round in such a way to reduce the burstiness of the produced schedule. However the worst-case delay bounds are still $O(N)$, and the matrix used to spread the service over the service round is not easy to recalculate when the flowset changes.

Stratified Round Robin (STRR) [**RP06**] groups flows on the basis of their weights, and uses a priority encoder to spread the service of each flow over each round; although the authors show a single-packet delay bound which does not depend on the number of flows, in the general case the WFI and the delay bound are still $O(N)$.

1.4. Timestamp Based Schedulers

The other big family of proportional schedulers is called *timestamp based* because its members assign timestamps to packets (or to flows) and try to serve them according to the order specified by these timestamps. The details vary from one scheduler to the other, but in general the schedulers belonging to this family internally track an ideal system (which can be a GPS or another reference system). Timestamps are, usually, the start and/or finish time in the ideal system. Completing the service of packets in the same order the ideal system would have served them allows this class of scheduler to obtain better service and delay bounds with respect to round robin schedulers.

Tracking the ideal system implies also defining a *virtual time* function, which tracks the service provided by the ideal system, and constitutes the reference point used to assign timestamps to newly arrived flows. Tracking the virtual time is one of the components, together with keeping flows sorted

according to their timestamps, that makes timestamp based schedulers more complex than round robin ones.

In particular, tracking the virtual time of a GPS reference system requires tracking the precise set of backlogged flows, to determine the rate at which virtual time increases. This was considered to have a linear cost until recently, in [**Val07**] an $O(\log N)$ algorithm to track the GPS virtual time was introduced. Several simplified virtual time functions have been proposed too, in the next section we will describe the one used in WF^{2}Q+.

One of the oldest timestamp based schedulers proposed in the literature was WFQ (Weighted Fair Queueing) [**DKS89**]; it serves packets according to their finish time in the corresponding GPS system. The Start Time Fair Queueing algorithm [**GVC97**] takes the other way round, serving packets according to their start times. Both algorithms need the exact tracking of the GPS virtual time, and both have a bursty behavior, due to the excessive deviation from ideal service. WF^{2}Q (Worst-case Fair Weighted Fair Queueing) [**BZ96**] introduced the eligibility constraint to WFQ, serving packets according to their finish time, but not before they arrive in the corresponding GPS system. WF^{2}Q has optimal deviation from the ideal service, but still needs exact virtual time tracking. WF^{2}Q+ (Worst-case Fair Weighted Fair Queueing Plus) [**BZ97**] was the first algorithm that achieved an optimal deviation from the GPS service using simplified virtual time tracking.

1.4.1. WF^{2}Q+. Here we outline the WF^{2}Q+ algorithm for a variable-rate system. See [**BZ97**, **SV97**] for a complete description. WF^{2}Q+ is a packet scheduler that approximates, on a packet-by-packet basis, the service provided by a work-conserving *ideal fluid system* which delivers the following, *almost* perfect bandwidth distribution over any time interval:

$$W^k(t_1,t_2) \geq \phi^k W(t_1,t_2) - (1-\phi^k)L. \tag{1}$$

The packet and the fluid system serve the same flows and deliver the same *total* amount of work $W(t)$ (systems with these features are called *corresponding* in the literature). They differ in that the fluid system may serve multiple packets in parallel, whereas the packet system has to serve one packet at a time, and is non preemptive. Because of these constraints, the allocation of work to the individual flows may differ in the two systems. WF^{2}Q+ has optimal B-/T-WFI and $O(\log N)$ complexity which makes it of practical interest.

WF^{2}Q+ operates as follows. Each time the link is ready, the scheduler starts to serve, among the packets that have already started in the ideal fluid system, the next one that would be completed (ties are arbitrarily broken). WF^{2}Q+ is a work-conserving on-line algorithm, hence it succeeds in finishing packets in the same order as the ideal fluid system, except when the next packet to serve arrives after that one or more out-of-order packets have already started.

This policy is efficiently implemented by considering, for each flow, a *flow virtual time* function $V^k(t)$ that grows as the *normalized amount of service* (i.e., the actual service divided by the flow's weight) received by the flow when it is backlogged. The algorithm only needs to know the values of $V^k(t)$ when the flow becomes backlogged, or when its head packet completes transmission in the ideal fluid system. So, each flow k is timestamped with these two values, called *virtual start* and *finish time*, S^k and F^k, of the flow. Using an additional *system virtual time* function $V(t)$, at time t_p when a packet

enqueue/dequeue occurs, WF^{2}Q+computes these timestamps as follows:

$$S^k \leftarrow \begin{cases} \max(V(t_p), F^k) & \text{on newly backlogged flow} \\ F^k & \text{on pkt dequeue} \end{cases} \qquad (2)$$
$$F^k \leftarrow S^k + l^k/\phi^k,$$

where $V(t)$ is the *system virtual time* function defined as follows (note that we assume $\sum \phi^k \leq 1$):

$$V(t_2) \equiv \max\left(V(t_1) + W(t_1, t_2), \min_{k \in B(t_2)} S^k\right). \qquad (3)$$

At system start-up $V(0) = 0$, $S^k \leftarrow 0$ and $F^k \leftarrow 0$.

Flow k is said *eligible* at time t if $V(t) \geq S^k$. The inequality guarantees that the head packet of the flow has already started to be served in the ideal fluid system. Using this definition, WF^{2}Q+ can be implemented as follows: each time the link is ready, the scheduler selects for transmission the head packet of the eligible flow with the smallest virtual finish time. Note that the second argument of the max function in (3) guarantees that the system is work-conserving.

The implementation complexity in WF^{2}Q+ comes from it having to accomplish three tasks:

(1) The computation of $V(t)$ from (3), which requires to keep track of the minimum S^k, and has $O(\log N)$ cost.
(2) The selection of the next flow to serve among the eligible ones, which requires sorting on F^k, and also has $O(\log N)$ cost at each step.
(3) The management of eligible flows as $V(t)$ grows. This is made complex by the fact that any change in $V(t)$ can render $O(N)$ flows eligible.

With some cleverness [**IS95**], an augmented balanced tree can be used to perform the latter two tasks together in $O(\log N)$ time.

CHAPTER 2

Budget Fair Queueing

MOST mainstream computer applications require moving data to/from disk devices. They range from local file copy to networked applications such as file transfer, Web, Video on Demand and Internet TV. Different applications have different requirements, from just high average disk throughput to guarantees on the completion time of each disk request.

Providing a predictable service to meet application requirements and at the same time achieving a high throughput is not an easy task. Three main difficulties can be highlighted. First, request *service time*, i.e., the time needed to serve a request once dispatched to the disk device, is dominated by seek and rotational latencies. These may be one or more orders of magnitude higher than the time needed to transfer (read/write) a sector once it is under the disk head. In addition, the transfer rate of the sector under the head may vary by a 30% or more as a function of the sector position on the disk. Finally, caching may significantly reduce the average request service time, but it increases its variance[1]. In the end, request service time and hence disk throughput are heavily influenced by the position and the *locality* of the requests (see below for issues related to disk internal queueing/reordering). An accurate knowledge of all the above disk parameters may make it possible, although not trivial, to soundly predict service times. However the second problem is that few if any of the existing controllers export these parameters. Modern controllers even hide disk geometry.

The third important problem is that, at least in Unix systems, most mainstream applications (including the ones mentioned at the beginning of this section) usually issue one request or one batch of requests at a time. Especially, they *block* until this request/batch has been completed. We denote the successive request or batch of requests issued by these applications as *synchronous*, for it/they can be issued only after the outstanding request(s) has/have been completed. As a consequence, if a disk scheduler delays the service of a request, it may actually delay the arrival of the successive synchronous request(s). Consider now a timestamp-based scheduler, i.e., a scheduler that (somehow) timestamps requests as a function of their arrival time and dispatches them to the disk in ascending timestamp order. Suppose that the scheduler always uses the *actual* arrival time of a request to timestamp it. Then, a delayed synchronous request may get a higher timestamp with respect to the one it would get not being delayed. This may 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

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

the service of (some of) the requests, the scheduler may force the application to issue requests at a *deceptively* low rate. If this anomaly occurs, the scheduler just fails to guarantee the reserved bandwidth or the expected request completion times, even to a greedy application.

A related problem is that the arrival of a synchronous request is further delayed by the fact that a minimum amount of time is needed for an application to handle a completed request and to submit the next synchronous one. From the disk device standpoint, the application is *deceptively idle* until it issues the next synchronous request. 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 (or even sequential) access.

Given the above problems, a simple way to guarantee a predictable and short request service time would be *over-provisioning*. For example only a few movies might be stored (and possibly replicated) on each of a large number of disks in a multimedia server. Unfortunately, over-provisioning entails higher economic costs. In fact, buying more resources is not the only source of additional costs: As reported in [**Law07**], the growing worldwide cost of powering and cooling computers was about half of the purchase expenditures in 2005. Moreover, since the purpose of over-provisioning is keeping disks underutilized, disk bandwidth would be wasted.

On the opposite end, the only option to improve disk utilization and meet the requirements of many competing applications is properly scheduling disk requests. With regards to the first goal, several algorithms—such as SCAN (Elevator), C-SCAN, LOOK or C-LOOK [**WGP94**]—can be found in the literature. This type of algorithms are often implemented also inside modern disk devices, which can prefetch and internally queue several requests, and then service them in the best order to boost the throughput[2].

Unfortunately, these algorithms do not take the deceptive idleness problem into account. As a consequence, as shown in [**ID01**] and confirmed by our experimental results, their performance may be quite low in the presence of synchronous requests. The problem is mitigated by the fact that, to preserve a high throughput, operating systems typically issue request(s) ahead, on behalf of processes deemed as issuing synchronous sequential read requests. However, read-ahead is usually not performed for non-sequential request patterns. The reason probably lies in that, on one hand, knowing beforehand the arrival time and the position of disk requests would be virtually impossible for most dynamic applications. On the other hand, deviations from the expected pattern would result in waste of CPU/memory/bus/disk usage and possible additional throughput loss. Accordingly, most Linux standard disk schedulers tackle the deceptive idleness by just not dispatching any other request to the disk device 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. Differently from prefetching, only disk time may be wasted. Although preventing disk internal queueing/reordering of requests, this simple disk idling approach has proven effective in providing a high throughput to mainstream applications as file and Web servers. This is the case for the Anticipatory disk scheduler [**ID01**], which can be seen as an extended non work-conserving version of C-LOOK that performs also disk idling (see [**Dis**] for

[2]The mechanism providing this feature is known as Native Command Queueing (NCQ) in SATA disks and Tagged Command Queueing (TCQ) in SCSI disks.

an extensive comparative analysis of the performance of several disk schedulers—in terms of aggregate throughput—as a function of different workloads and also in presence of internal queueing).

However, to meet the requirements of most types of applications, guarantees must also be provided on bandwidth distribution and request completion times. In this respect, the common problem of algorithms aimed only at maximising 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 (although additional age-based policies, as in Anticipatory, can be integrated with the base scheduling algorithm to mitigate the problem). In contrast, many schedulers, as e.g., SCAN-EDF [**RW93**], SATF-DAS [**RP03**], JIT [**MJR97**], Hybrid [**RV04**], YFQ [**BBG+99**], pClock [**GMV07**], adaptive DRR [**GMUV07**], CFQ [**Axb**] and adaptations of SFQ [**JCK04**] have been proposed to provide more control on request completion time and/or on bandwidth distribution, while at the same time trying to keep the throughput high.

A first practical difficulty with most of these schedulers stems from the fact that, except for round robin schedulers, the accuracy of the provided service guarantees depends on the accuracy in the knowledge of the disk physical parameters, as detailed in Section 2.3. These parameters, in turn, are usually extracted by running special disk benchmarks. Unfortunately, due to the previously mentioned issues, the values returned by benchmarks are necessarily approximated. In the end, discrepancies between actual and estimated values affect both guarantees on request completion time (which is unavoidable), and guarantees on throughput distribution, which is avoidable, as shown in this paper.

However, the most serious problem is that, according to the authors' knowledge, none of the schedulers proposed in the literature so far takes either the delayed arrival or the deceptive idleness problem into account. According to the previous arguments, in presence of synchronous requests this fact may cause loss of disk throughput, as well as complete violation of the expected service guarantees in case of timestamp-based schedulers. For similar reasons, guarantees are likely to be violated with any scheduler if the disk device performs internal queueing, as discussed in more detail in Section 2.3. Our experimental results thoroughly confirm the expected loss of both guarantees and disk throughput for all the analyzed timestamp-based schedulers.

In contrast, a non timestamp-based scheduler should not suffer from these problems, provided that it adopts at least some mechanism to handle deceptive idleness. This is the case for round robin schedulers. Unfortunately, as discussed in Section 2.3 and experimentally shown in Section 2.4, they exhibit a higher delay/jitter in request completion times than the scheduler we propose.

Proposed Solution. In this paper we propose a new *proportional share* timestamp-based disk scheduling algorithm, called Budget Fair Queueing (BFQ). In BFQ each application is reserved the desired fraction (share) of the disk throughput. BFQ assigns to each backlogged application a *budget*, measured in number of sectors to transfer. Once selected, an application gets exclusive access to the disk, until it consumes either its budget or its backlog. BFQ adopts the previously described disk idling approach to handle deceptive idleness, and some simple additional timestamping rules to deal with delayed arrivals.

The budgets assigned to an application vary over time as a function of its behaviour, but they are never larger than a *user-configurable system-wide* maximum value B_{max}. Budgets are internally scheduled according to a slightly extended version of WF^{2}Q+ [**BZ97**], which we introduced in Section 1.4.1. Thanks to this internal scheduler, called Budget-WF^{2}Q+ (B-WF^{2}Q+), and under the constraint of

serving applications budget-by-budget, BFQ guarantees to each application the minimum possible *lag*, over *any* time interval, with respect to its *reserved service*, i.e., with respect to the minimum amount of service, measured in number of sectors transferred, that the application should receive according to its reserved share. A loose upper bound to this lag is $3B_{max}$.

On one end, the scheduling policy of BFQ can be fine-tuned by setting both B_{max} and several other configuration parameters. On the opposite end, a simplified interface is provided for users not concerned with low-level details. If this interface is used, all the parameters except B_{max} are set to system-dependent default values (more precisely the same typical values used by the Linux disk schedulers). The user just sets either directly B_{max}, or a *throughput boosting level* parameter varying from 0 to 1. In the latter case, the back-end of the interface takes care of setting B_{max} accordingly, between a system-dependent minimum and maximum value (details in Section 2.2.3). Summarizing, BFQ can be either accurately or very easily configured to achieve the desired trade-off between throughput boosting and maximum per-application lag, ranging from unlimited budgets (best throughput but no guarantees) to request-by-request proportional share (most accurate guarantees but poor throughput). These *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, i.e., the request pattern causing the lowest possible aggregate throughput (see Section 2.2.3). Then, worst-case time guarantees can be computed as function of the aggregate throughput (they are inversely proportional to it) using a simple formula.

Contributions. BFQ enjoys the following properties:

(1) Strong guarantees on throughput distribution: Thanks to the fact that the internal B-WF^{2}Q+ scheduler works in the service and not in the time domain (see Section 2.1.4), each application is guaranteed its reserved share of the aggregate disk throughput, under every work condition and regardless of the value of the disk physical parameters.
(2) Practicality: Thanks to the same characteristic of the internal scheduler, BFQ is defined without ever using disk physical parameters, hence it can be implemented—and its time guarantees assessed—without knowing the values of the disk physical parameters. The only information needed is the locality of the expected request pattern. In this respect, it is worth noting that, because of the previously mentioned dominant influence of request locality on service time, always considering the very worst-case request service time for computing service guarantees is overly pessimistic. Hence, with any scheduler, there is no possibility to provide practical time guarantees without knowing at some degree the (locality of the) expected request pattern. This is the only requirement to perform the benchmarking procedure mentioned in the previous subsection and hence to provide time guarantees with BFQ.
(3) Flexibility: Short-term guarantees can be traded off for disk throughput in a very simple way, whereas long-term guarantees are unconditionally provided.
(4) Compliance with both the delayed arrival and the deceptive idleness problems: Service guarantees and disk throughput are not affected by these problems.

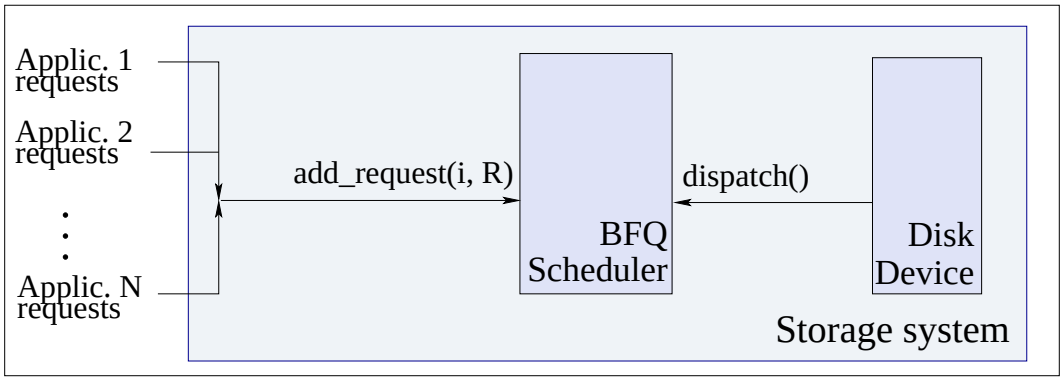


FIGURE 1. Reference Storage System.

(5) Efficiency: As shown in Section 2.1, BFQ has $O(\log N)$ cost per request insertion or dispatch, where N is the number of competing applications. This cost boils down to $O(1)$ if approximate implementations [**SBZ99**] of B-WF^{2}Q+ are adopted.

Finally, several implementations of BFQ for the Linux kernel are currently available[3]. In this paper we evaluated, through experimental results, the effectiveness of the version for the 2.6.21 kernel in achieving high throughput as well as guaranteeing the expected bandwidth distribution and per-request completion times to file transfer, Web server and video streaming applications.

Organization of the Chapter. In Section 2.1 we describe BFQ, whereas in Section 2.2 we report its service properties in the sector and time domains. In Section 2.3 we provide a brief survey of related work. Finally, in Section 2.4 we compare the performance of BFQ against the one of research and production schedulers.

2.1. Budget Fair Queueing

Consider the *storage system* shown in Figure 1. It is composed of a disk device and the BFQ scheduler. The former contains a disk, which we model as a sequence of contiguous fixed size sectors (chunks of bytes), each identified by its *position* (index) in the sequence. The disk device services two types of *disk requests*, respectively the reading and the writing of a set of contiguous sectors from/to the disk. 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 stand for the possible entities that can compete for disk access in a real system, as, e.g., *threads* or *processes* in our Linux implementation). The meaning of the notations hereafter introduced is also summarized in Table 1. The i-th application issues its j-th 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. The BFQ scheduler enqueues R_i^j in its internal data structures.

[3]The latest available versions at the time of this writing are for 2.6.25 and 2.6.27, and the former is in public testing before being possibly included in one of the next releases of the kernel [**BFQ**].

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
B_i^l	l-th budget assigned to the i-th application
T_{FIFO}	Queueing time after which (queued) requests must be served in FIFO order

We define as *size* L_i^j of R_i^j the number of sectors to transfer (read/write), and as *position* of R_i^j the position of the first of these sectors. 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 i-th 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 as *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 (as shown in detail in Section 2.1.1) an application is denoted as *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

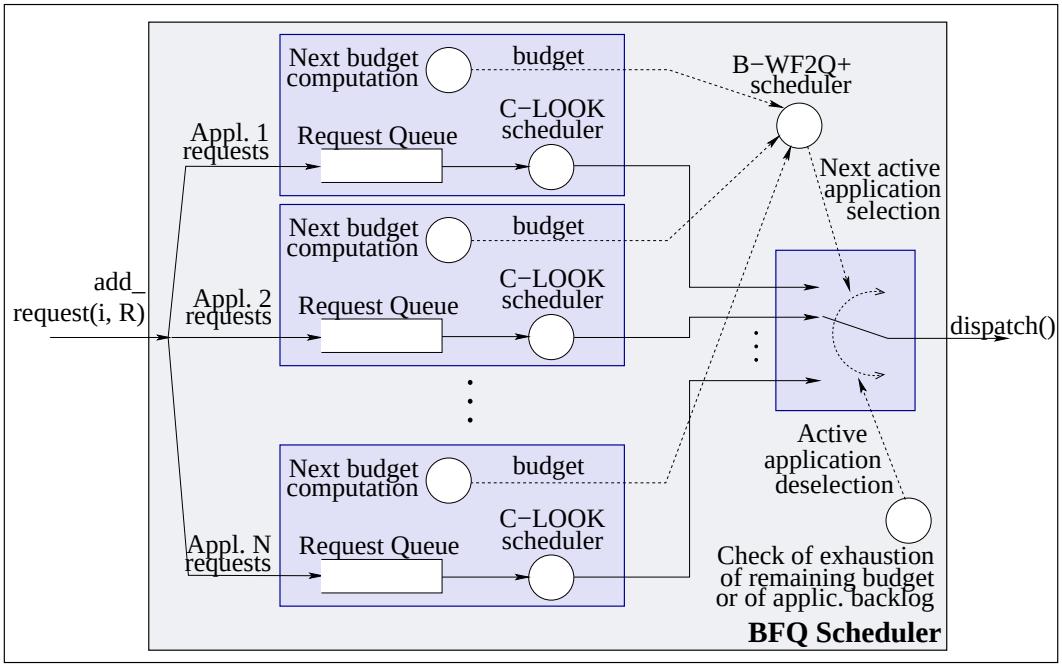


FIGURE 2. BFQ Logical Scheme.

Section 2.2, whereas hereafter we assume that applications do enjoy this property, and we show how BFQ (more precisely B-WF^{2}Q+) conceals delayed arrivals for these applications.

Each application has a fixed weight ϕ_i assigned to it. Without losing generality, we assume that $\sum_{q=1}^{N} \phi_q \leq 1$. Hence a new application with a given weight can be *admitted*, i.e., added to the set of applications competing for the disk, if and only if this inequality holds also after adding the weight of the application. 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 i-th 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 i-th application (ideally) 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 $c_i^j > \overline{c}_i^j$.

As shown in the next two subsections, describing the main algorithm of BFQ, both the Cyclical LOOK (C-LOOK) and the B-WF^{2}Q+ schedulers are used inside BFQ. C-LOOK services requests in ascending position order, starting over with the one at the lowest position when all the requests are *behind* the last served one. B-WF^{2}Q+ is described in detail in Section 2.1.4.

2.1.1. General Description. The logical scheme of the BFQ scheduler is depicted in Figure 2. Differently from Figure 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 in 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 l-th budget assigned to the i-th application. At system start-up, all applications are assigned the same *default* budget B_i^0, whereas, during system operation, the next budget of each application is computed as a function of its behaviour (see the comments to the pseudocode in the next subsection). Disk access is granted to one application at a time, denoted as the *active application*. When the new active application is selected, its current budget, say B_i^l, 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, possibly leaving the disk idle. As shown in detail in the next subsections, this wait allows the deceptive idleness problem to be effectively dealt with for applications enjoying the SI property. If the application issues no new request before the timer expiration, it is deemed idle. The active application is exclusively served until either the remaining budget is exhausted, i.e., 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. The next active application is then chosen by B-WF^{2}Q+, which schedules applications as a function of their budgets (see Section 2.1.4).

The order in which the requests are extracted from the queue of the active application depends on a user-configurable T_{FIFO} parameter. When the next request of the i-th 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, otherwise the oldest request is picked. This is one of the parameters that allow the desired trade off between guarantees on per-request completion time and throughput boosting to be set.

The last important configuration parameter considered in this introductory subsection is the system-wide maximum *time budget* T_{max} (possibly automatically computed, see Section 2.2.3). 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[4]. This additional mechanism prevents any excessively *seeky* application from over-lowering the disk throughput. Hence it guarantees practical bandwidths and delays (which are inversely proportional to the aggregate throughput, as shown in Section 2.2.2) to *well-behaved* applications. On the contrary ill-behaved applications virtually receive no service guarantees. As the rest of the scheduler and the form of the guarantees are completely independent of this additional mechanism, for brevity, its simple implementation is not reported in the pseudo-code.

2.1.2. Main Algorithm. The main BFQ algorithm is shown in Figure 3 using pseudocode. The function `add_request()` first inserts the new request R in the application queue, then, if there is 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. On the contrary, if the (active) application was waiting for the arrival of its next request, the timer is canceled.

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. On the contrary, if the active application has

[4]In the last publicly available release of BFQ a simple heuristic even allows the scheduler to detect in advance if an application is consuming its budget to slowly. See the code for details [**BFQ**].

not enough remaining budget for its next request, its budget is deemed as exhausted, and the application is deselected (line 35). Moreover, through the call to the function `b-wf2q+_update_vfintime()` (line 29, described in the next subsection), the application timestamps 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 32–33). Finally, the application is enqueued into B-WF^{2}Q+ with its new budget (line 34). Then, if no application is active (lines 37–41), the next active application is picked (and removed) from the B-WF^{2}Q+ scheduler, and the budget of this application is assigned to the remaining budget counter. The next request R to serve is extracted from the queue of the active application and the remaining budget counter is decremented (lines 42–47).

Finally, if the queue of the active application becomes empty, the timer is set to the current time plus T_{wait} (lines 48–49). Waiting for the next request of the active application prevents BFQ from switching to a different application if the former is deceptively idle and enjoys the SI property. This allows BFQ not to break a possible profitable sequence of close or sequential accesses (performance gains and trade offs are discussed in detail in Section 2.2). It is also instrumental in concealing delayed arrivals, as discussed in the next subsection. If no new request is issued by the application before timer expiration, the function `timer_expiration()` (lines 57–65) gets called: The application is declared idle, and a new budget equal to the previously consumed one is assigned to it. Finally, the timestamps of the application are properly updated to account only for the actual amount of service it received (see the next subsection).

Of course, the simple linear increase/instantaneous decrease scheme used in updating budgets is only one of the possible options. In our experiments it showed good performance in terms of both aggregate throughput and short term guarantees. For simplicity, many low level details have been omitted here. The interested reader is referred to [**BFQ**]. Before leaving this subsection, 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 easily replaced with different schedulers (Figure 2). Hence, different trade-offs among type of guarantees, computational cost and throughput boosting degree can be achieved.

2.1.3. Disk Weighted Fair Queueing. In this subsection we provide a brief survey of the main concepts behind the original WF^{2}Q+ algorithm, (re)formulated in the disk scheduling domain (see Chapter 1, [**BZ97**] and [**SV97**] for details). These concepts are the basis for B-WF^{2}Q+, which is then described in detail in the next subsection.

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,

```
1  active_appl = none ; // reference to the currently active application
2  remaining_budget = 0 ; // remaining budget of the active appl.
3
4  // input: application index, request issued by the application
5  add_request(int i, request R)
6  {
7    appl = applications[i] ; // reference to the i-th application
8
9    // insert R in the application queue
10   enqueue(R, appl.queue) ;
11
12   if (appl.queue.size == 1) { // the queue was empty
13     if (appl != active_appl)
14       b-wf2q+_insert(appl) ; // Sec.2.1.4
15     else   // appl is the active application
16       if (waiting_for_the_next_req()) // deceptive idleness
17         unset_timer() ; // next req arrived: stop waiting
18   }
19 }
20
21 // output: request to serve
22 request dispatch()
23 {
24   if (all_applications_are_idle() OR waiting_for_the_next_req())
25     return no_request ;
26
27   if (active_appl != none AND
28       remaining_budget < C-LOOK_next_req(active_appl.queue).size) {
29         b-wf2q+_update_vfintime(active_appl,
30             active_appl.budget - remaining_budget) ;
31
32         if (active_appl.budget + BUDG_INC_STEP <= active_appl.max_budget)
33             active_appl.budget += BUDG_INC_STEP ;
34         b-wf2q+_insert(active_appl) ;
35         active_appl = none ;
36   }
37   if (active_appl == none) {
38     // get and extract the next active appl. from b-wf2q+ (Sec. 2.1.4)
39     active_appl = b-wf2q+_get_next_application() ;
40     remaining_budget = active_appl.budget ;
41   }
42   // get and remove the next request from the queue of the active application
43   next_request = dequeue_next_req(active_appl.queue) ;
44
45   // for simplicity at this point we assume that next_request.size ≤
46   // remaining_budget, see the code [BFQ] for details
47   remaining_budget -= next_request.size ;
48   if (is_empty(active_appl.queue))
49       set_timer(T_wait) ; // start waiting for the next request
50
51   // account for this service in b-wf2q+
52   b-wf2q+_inc_tot_service(next_request.size) ;
53
54   return next_request ;
55 }
56
57 timer_expiration()  // called on timer expiration
58 {
59   active_appl.budget =
60     active_appl.budget - remaining_budget ;
61   b-wf2q+_update_vfintime(active_appl,
62                 active_appl.budget) ; // Sec. 2.1.4
63   active_appl = none ; // no more the active application
64   // dispatch() will take care of selecting the next active application
65 }
```

FIGURE 3. BFQ Main Algorithm.

as opposed to the real system, i.e., to the storage system in Figure 1. The ideal system is used as a reference because it distributes the total service among applications is such a way to guarantee to each application a bounded lag, over any time interval, with respect to its reserved service. 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 the Section 2.2, 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 timestamping 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 the pending batch at time t. They are computed as a function of a common *system virtual time* function $V(t)$ (see [**BZ97**] or [**SV97**]). Although it seems a contradiction in terms, the unit of measure of the virtual time is not the time, but the service (see Section 2.2 for details). Virtual start and finish times do not vary over time and can be immediately computed as a function of $V(t)$ (and independently of any disk parameter) when an application is inserted into the scheduler.

2.1.4. B-WF^{2}Q+. The B-WF^{2}Q+ algorithm is shown in Figure 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 request arrival time, using the same formulas as in WF^{2}Q+ [**BZ97**]. On the contrary, if the inserted application is the active one (which, according to Figure 3, means that it is being enqueued in B-WF^{2}Q+ after a deactivation), the application is timestamped as if its next request to serve had arrived upon the (ideal) completion time of the last request of the last finished batch in the ideal system. In other words, the arrival time of this request is fictitiously *shifted backwards* to no later than the ideal completion time of the previous request. Hence a possible delayed arrival has no negative impact on the timestamps of the application. Moreover, this backward shift does not affect the worst-case guarantees provided to the other applications, as the latter are independent of the exact arrival time of this request.

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 26–27). In case there are quasi-backlogged applications, but no one is eligible (lines 22–25), $V(t)$ is *pushed-up* to the minimum virtual start time among *all* the quasi-backlogged applications (the latter coincide with the applications enqueued in B-WF^{2}Q+ when `b-wf2q+_get_next_application()` is invoked). Since an application

```
1  V = 0 ; //system virtual time
2
3  //input: the application to insert
4  b-wf2q+_insert(appl)
5  {
6    if (appl != active_appl) //then an asynchronous req arrived
7      appl.S = max(V, appl.F) ; //use actual arrival time
8    else //in this case the active appl. is being deactivated
9      //to preserve guarantees in case of delayed arrival, timestamp
10     //the application as if the next request to serve arrived at the same
11     //time as the last request of the just terminated batch
12     appl.S = appl.F ;
13   appl.F = appl.S + appl.budget / appl.weight ;
14   //add the application to the internal data structure
15   set_of_enqueued_appl.insert(appl) ;
16 }
17
18 //extract and return the next application to serve
19 b-wf2q+_get_next_application()
20 {
21   if (there_is_no_eligible_appl())
22     //certainly there are enqueued applications, otherwise this
23     //function does not get called (see function dispatch)
24     V = set_of_enqueued_appl.minS() ;
25   next_appl = eligible_appl_with_minF() ;
26   set_of_enqueued_appl.remove(next_appl) ;
27   return next_appl ;
28 }
29
30 //called upon each request dispatch
31 b-wf2q+_inc_tot_service(service)
32 {
33   V = V + service ;
34 }
35
36 //update application virtual finish time
37 b-wf2q+_update_vfintime(appl, received_service)
38 {
39   appl.F = appl.S + received_service / appl.weight ;
40 }
```

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

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 [**BZ97**] (however, as shown in Section 2.1.1 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 i-th application be one of the applications that are idle in the real system at time $\overline{t}$ when a jump is performed. Suppose for a moment that a delayed synchronous request R_i^j arrives after time $\overline{t}$. Since the i-th application is not taken into account in computing $V(\overline{t})$, the jump would be conceptually incompatible with a fictitious backward shift of the arrival of R_i^j to (or before) time $\overline{t}$. Hence it is easy to show that it would not be possible to conceal this delayed arrival without violating the guarantees of the i-th application. Fortunately, if the i-th application enjoys the SI property, it is not possible that $a_i^j > \overline{t}$, 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 32-35). 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 step-wise increment of $V(t)$, the application may be timestamped 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 2.2.

The last important difference between B-WF^{2}Q+ and WF^{2}Q+ is that, differently from a packet system, an application may not use all of its budget. This difference affects only time guarantees, as shown in Section 2.2.2. Here we highlight only 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 38–41) 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 [**SBZ99**], depending on whether exact or approximate timestamps are used. Since all the other operations in Figure 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 subsection does not contain many of the details of the complete version. The latter, e.g., autonomously adapt 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 [**Val05**]. 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 also after replacing ϕ_i with $\frac{\phi_i}{\Phi_{TOT}}$ in the following inequalities.

2.2. Service Properties

In this section we show the service properties of BFQ, in both sector (bandwidth distribution) and time domains. We also show how to achieve the desired trade-off between guarantee 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 presence of delayed arrivals. From Section 2.1 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, T_{wait} has an important impact on the system performance. It provides significant throughput boosting in presence of deceptive idleness, but only if its value is in the order of the seek and rotational latencies [**ID01**], 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 intensive use of the disk device and phases during which they rarely access it. Fortunately, even for the above mentioned beneficial low value of T_{wait}, during the former phases the SI property holds for the majority of the mainstream applications: File copy, file transfer, audio/video streaming (where request arrivals are time-driven) and so on. 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.

2.2.1. Sector Guarantees. 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 [**BZ97**]. 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 2.1). The following theorem holds.

THEOREM 2.1. *For any time interval* $[t_1, t_2]$ *during which the* i*-th application is continuously quasi-backlogged, BFQ guarantees that:*

$$\phi_i \cdot W(t_1, t_2) - W_i(t_1, t_2) \leq B_{max} + B_{i,max} + L_{max}. \tag{4}$$

The right hand side in (4) 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 2.1 is basically an extension of the proof of the Bit-WFI of WF^{2}Q+ [**BZ97**], and can be found in Appendix A, while 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 i-th applications 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 i-th application for at most $B_{i,max}$ sectors with respect to the ideal system at time t_1. Hence 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 minimum completion time guaranteed in the ideal system to the previous request

R_i^{j-1}. Hence, on the opposite end, if 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 may have served at most L_{max} sectors.

It is important to note that the rightmost term in (4) does not grow with the time or the (total) amount of service, hence the long term bandwidth distribution is *unconditionally* guaranteed. Furthermore, (4) 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 2.2.3. 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.

2.2.2. Time Guarantees. The following theorem is the starting point for computing BFQ time guarantees. Let $t_1 \leq a_i^j$ be a generic time instant such that the i-th 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 i-th 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 i-th application, be set to its average value in the real scenario. Finally, thanks 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 2.2. *Given a request R_i^j, let $t_1 \leq a_i^j$ be a generic time instant such that the i-th 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 i-th application during $[t_1, a_i^j + T_{FIFO})$ plus L_i^j. The following inequality holds:*

$$c_i^j - t_1 \leq \frac{Q_i(t_1^-) + A_i(t_1, a_i^j + T_{FIFO}) + (B_i^l - L_i^j) + B_{i,max}}{\phi_i T_{agg}} + \frac{B_{max} + B_{i,max} + L_{max}}{T_{agg}}, \tag{5}$$

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

As before, the proof of this theorem (which can be found in Appendix A) is just an extension of the proof of the Time-WFI of WF^{2}Q+. Here we discuss the terms in (5), and then how to use (5) to provide actual time guarantees.

The right hand side of (5) can be rewritten as $\frac{1}{\phi_i T_{agg}} \cdot \left(Q_i(t_1^-) + A_i(t_1, a_i^j + T_{FIFO})\right) + d_i^j$. 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. In contrast d_i^j represents the *worst-case delay* with respect to the ideal worst-case completion time. With regards to the first component, note that, if the i-th application issues only synchronous requests, the latter are 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 timestamped (and scheduled) as a function of L_i^j, but as a function of B_i^l. Hence, intuitively, 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}, $B_{i,max}$ and L_{max} terms can be explained using the same arguments as for the same terms in (4).

We now discuss how to use (5) for providing time guarantees. First, the aggregate throughput T_{agg} must be known at some extent to assess the actual time guarantees. 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 2.2.3, basing upon (5), 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 actual completion times depends on how accurately T_{agg} itself is known (worst-case or average value, variance, confidence interval, ...). Then, for each application (and within the limits imposed by the worst-case delay), the desired request completion time can be guaranteed by setting its weight. In general, 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.

Finally, as an example of use of (5) for soft real-time applications, such as e.g., audio or video streaming, consider a periodic or sporadic application issuing requests with a size of at most Q_i sectors, and with a period or minimum inter-arrival time of P_i seconds. Since $T_{agg} \cdot P_i$ sectors are transferred during P_i, it follows that, to meet the bandwidth requirements of the application, it is enough to set $\phi_i = \frac{Q_i}{T_{agg} \cdot P_i}$ (of course, as stated in Section 2.1, for the application to be admitted, the resulting sum of the weights $\sum_{q=1}^{N} \phi_q$ must still be no higher than 1). Notably, according to the reserved service of the i-th application, 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 subsection, the $B_{i,max}$ component is absent from (5). In the end, the application must tolerate a worst-case delay (jitter)

$$d_{i,max} \equiv \max_j d_i^j \leq \frac{B_{i,max} - L_{i,min}}{\phi_i T_{agg}} + \frac{B_{max} + L_{max}}{T_{agg}}, \tag{6}$$

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

To show possible values of the bounds (5) and (6) in a real system, and to demonstrate the feasibility of interactive and soft real-time applications with BFQ, both a Web and a Video-on-Demand service are considered in Section 2.4. Finally, note that only short-term guarantees are affected by $B_{i,max}$, B_{max}

and T_{FIFO}, thus the requirements of file read/copy/transfer applications can be effectively fulfilled by simply configuring BFQ for maximum throughput.

2.2.3. 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 presence of asynchronous requests. 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). Finally, the effect of T_{max} is strictly related to the workload characteristics.

However, according to (4) and (5), in addition to the disk throughput, all these parameters directly or indirectly influence guarantees. In the end, 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.

In addition, for a user not interested in full control over all the parameters and in the resulting tuning complexity, BFQ also provides a simplified interface. If this interface is used, T_{FIFO} is set to the default value used in the system (i.e., the default value used by the other schedulers in the kernel, 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} are 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 [**BFQ**] and guarantees a very high throughput, as shown in Section 2.4. 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 2.2.2). We used the simplified interface in our experiments.

However, 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 pattern. Such a pattern may be defined as a function of the information available on the expected request pattern. If very little information is available, the following conservative worst-case pattern may be used to evaluate the expected minimum aggregate throughput. Suppose that only two pieces of information are available:

(1) The maximum span $P_{last} - P_{first}$ of the positions of the requests issued by the applications.
(2) The minimum size S_{min} of the (portions of the) files that will be interested by sequential accesses.

After placing two files with size S_{min} at the maximum possible distance in $[P_{first}, P_{last}]$, a simultaneous sequential read of the two files may be performed. As confirmed also by our experiments, the resulting aggregate throughput is likely to be a lower bound to the aggregate throughput for file transfer, web server and mixed video streaming/file transfer workloads.

2.3. Related Work

Existing solutions for providing a predictable disk service can be broadly classified into scheduling algorithms and frameworks. The former can in turn be divided into three groups:

(1) Real-time disk schedulers [**DS94**], such as, e.g., Priority SCAN (PSCAN), Earliest Deadline SCAN, Feasible Deadline SCAN (FD-SCAN) [**KGM95**], SCAN Earliest Deadline First (SCAN-EDF) [**RW93**], Rotational-Position-Aware disk scheduling based on a Dynamic Active Subset (SATF-DAS) [**RP03**] and Just-In-Time Slack Stealing (JIT) [**MJR97**].
(2) Proportional share or *bandwidth reservation* timestamp-based disk schedulers (also known as *fair-queueing* schedulers), such as, e.g., Yet Another Fair Queueing (YFQ) [**BBG+99**], Hybrid [**RV04**], adaptations of SFQ [**JCK04**] and pClock [**GMV07**] (which is actually more general than all the others, in that it allows application requirements to be expressed in terms of throughput, latency and maximum burst size).
(3) Proportional share round robin disk schedulers, such as, e.g., adaptive DRR [**GMUV07**] and Complete Fair Queueing (CFQ) [**Axb**].

Finally, examples of frameworks for providing QoS guarantees are Cello [**SV98**], APEX [**KL05**], PRISM [**RWW05**] and Argon [**WAEMTG07**]. It is worth mentioning also real-time operating systems such as the Dresden Real-Time Operating System (DROPS) [**RP03**] and RT-Mach [**MJR97**], and Real-Time Database Systems (RTDBS), which are architectures for performing database operations with real-time constraints [**KGM95**].

We observe that 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, in the following subsections we focus only on each of the above listed classes of scheduling algorithms. In addition, in Section 2.3.2, we also show the loss of guarantees related to disk internal queueing.

2.3.1. Real-Time Schedulers. Real-time schedulers [**DS94**, **RW93**] associate a deadline to each request, hence they usually start from an Earliest Deadline First (EDF) [**LL73**] schedule and reorder requests with the goal of reducing seek and rotational latency without violating deadlines.

Real-time disk schedulers are quite similar in principle, hence, for ease of exposition, to highlight the problems that all of them share we describe only SCAN-EDF here. It is one of the most flexible real-time scheduler in trading guarantee granularity for throughput boosting. SCAN-EDF 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). Real-time applications are assumed to be periodic (or sporadic) and to have successfully reserved the service time they need over each period (or over each minimum inter-arrival time). Hence (as for any real-time scheduler) knowing the worst-case service time of each request is critical to assess the feasibility of the application requirements. Request service times are usually computed as a function of the disk physical parameters. Hence, as anticipated in the introduction, differently from BFQ, these parameters need to be (accurately) known to provide service guarantees.

In a pure real-time system (worst-case) arrival times are known beforehand and are independent of the service, hence they cannot be delayed. However, real-time bandwidth servers can be built on top of EDF-based schedulers. If the former are used to distribute the bandwidth among applications issuing synchronous requests in a general-purpose system, guarantees may be violated as a consequence of delayed arrivals. To investigate the problem, we implemented a slightly extended version of SCAN-EDF in the 2.6.21 Linux kernel [**BFQ**]. In this implementation, 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 (absolute) deadline of each request is then computed as the sum of the relative deadline and the maximum between the request arrival time and the deadline of the previous request of the same application. The resulting algorithm can be seen as a simple EDF-based bandwidth server. Suppose that the i-th 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 bandwidth equal to L_i/P_i.

Unfortunately, if the application issues synchronous requests this guarantee is violated. First, there is the deceptive idleness problem. It takes a while for the request R_i^{j+1} to arrive after the completion of the request R_i^j at time c_i^j. Hence, even if the not yet arrived request R_i^{j+1} would have a lower deadline than all the currently pending requests, at time c_i^j the scheduler starts serving the request of another application. To deal with the deceptive idleness, we further extended SCAN-EDF as follows: After the completion of the last request of an application, the disk is kept idle until either a new request of the just served application arrives, or a configurable T_{wait} time interval elapses. However, this resulted to be insufficient to preserve guarantees, for the following reason. Deadlines may be missed, mainly because of the non-preemptability of request service, hence the arrival of synchronous requests may be delayed. As confirmed by the experimental results (Section 2.4) (and as it would happen for any real-time bandwidth server with similar deadline computation rules), the expected guarantees are violated because the absolute deadlines of the delayed requests are computed as a function of their actual arrival time.

To avoid delayed arrivals, relative deadlines may be fictitiously reduced to guarantee the actual deadlines to be met. Unfortunately this approach causes bandwidth waste and loss of disk throughput (further modifications to SCAN-EDF are out of the scope of this paper). 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. Unfortunately the request size is not under the control of the scheduler in the Linux kernel. On the contrary, as a simple way to control both the extension of the deadlines and the probability that close deadlines are treated as equal, our implementation 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 $n \cdot \Delta$. As shown in Section 2.4, provided that the arrival of synchronous requests is waited for, increasing the value of Δ allows the throughput to be effectively boosted.

2.3.2. Proportional Share Timestamp-based Schedulers. To show the problems shared by all the proportional share timestamp-based schedulers mentioned at the beginning of this section, in this subsection we refer to YFQ, as its simple policy also allows us to easily highlight the problems related with disk internal queueing. YFQ dispatches requests to the disk device in *batches* of BT_{size} requests, where BT_{size} is a user-configurable parameter. The requests in a batch may be served with the desired throughput boosting algorithm. Before the next batch is served, all the requests in the current batch must be completed. The same batch-by-batch service occurs in case of disk internal queueing.

Differently from BFQ, YFQ directly targets proportional time allocation instead of sector allocation. To this purpose YFQ charges each request for its expected service time, given by the sum of the sector transfer time and the average latency it is presumed to incur. The latter quantity is estimated in a heuristic way. As a consequence, differently from BFQ, the accuracy of disk throughput distribution depends on the accuracy in estimating request service times.

To compare it against BFQ, we implemented YFQ in the 2.6.21 Linux kernel [**BFQ**]. This implementation extends the basic YFQ algorithm to handle deceptive idleness in the same way as illustrated for SCAN-EDF. Each batch is served in C-LOOK order. Batch overlapping, one of the enhancements proposed by the authors of YFQ to increase disk throughput, is performed as well (see the code [**BFQ**] for further details). However, when asked for the next requests to serve, YFQ dispatches an entire batch, which in general contains requests issued by different applications. First, this reduces the possibility of close or sequential accesses of the same applications. Second, even if an application has a (much) higher weight than the others but it issues synchronous requests, then no more than one request of the application will be served for each batch. It is worth noting that both problems are not related to the specific policy by which the requests to insert in a batch are chosen, but are inherent to the batch-by-batch service scheme. As a consequence, also disk internal queueing mechanisms and any scheduler based on the same service scheme unavoidably suffers from both problems.

In addition, as any of the proportional share timestamp-based schedulers mentioned in this section, YFQ timestamps requests as a function of their actual arrival times. Hence it may fail to provide the expected guarantees as a consequence of the delayed arrival problem. In fact, as confirmed by the experimental results in Section 2.4, independently of whether or not synchronous requests are waited for, YFQ (and hence these schedulers) fail both to guarantee the desired bandwidth distribution and to provide a high throughput in presence of synchronous requests.

2.3.3. Proportional Share Round Robin Schedulers. To show the common characteristics of this class of schedulers here we describe in detail CFQ, which is a proportional share disk scheduler that grants disk access to each application for a given time-slice T_{slice}, a system-wide user-configurable parameter. Slices are scheduled according to a round robin policy. The time-based allocation of the disk service in CFQ 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 towards 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 (the same problem occurs in the Argon framework, as the same time-based approach as CFQ is adopted). It is important to note that BFQ owes to CFQ the idea of exclusively serving each

application for a while. But BFQ provides strong guarantees on bandwidth distribution because the assigned budgets are measured in number of sectors. Moreover, as any round robin scheduler, CFQ is characterized by an $O(N)$ worst-case jitter in request completion time, where N is the number of competing applications. On the contrary, thanks to the accurate service distribution of the internal B-WF^{2}Q+ scheduler, BFQ exhibits $O(1)$ jitter according to (5) with respect to the number of applications. Especially, in Section 2.4, after configuring BFQ so as to let a maximum budget be consumed in about a default CFQ time-slice (and hence to let the rightmost term in (5) be comparable to this time-slice), a quantitative evaluation of the consequences of this different short term guarantees and of the above mentioned unfairness of CFQ is reported.

2.4. Experimental Results

In this section we report the results of our experiments with BFQ, SCAN-EDF, YFQ (more precisely with our implementation of these three schedulers), CFQ, C-LOOK, and Anticipatory (AS for brevity) in the Linux 2.6.21 kernel. According to what is said in Section 2.3, the experimental results for the latter five schedulers actually show the common problems of all the schedulers reported in that section. 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. Due to 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 [**BFQ**].

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, $\sim$ 35% 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 in all the experiments. 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, the last one covering the inner part. 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, and the execution of each experiment with a given set of values of the parameters was interspersed between the execution of experiments with different values of the parameters). The minimum, maximum, and mean value, together with its associated 95% 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 semi-width of the 95% confidence interval for v. The simplified interface (Section 2.2.3) was used to set (only) the maximum budget B_{max} with BFQ.

The first set of experiments was aimed at estimating the worst-case aggregate throughput guaranteed by each scheduler in presence of simultaneous sequential reads. 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 5 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 $\{512, 1024, 2048, 4096, 8192, 16384\}$ 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 $\{128, 256, 512, 1024\}$ MB. Of course, T_{wait} is implicitly 0 ms in C-LOOK, whereas it was set to 4 ms in AS. Moreover the AS parameters were set so as to deactivate any additional fairness policy. 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.

With any scheduler, the lowest throughputs were achieved in case of 2, 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 a MB/s). For this scenario, the following table 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:

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	

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 wait for the arrival of the next synchronous request of the just served application. It is easy to see that, for $B_{max} = 16384$ 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 1 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 transfer rate, it is easy to see that less than 16384 sectors can be read in 100 ms (which is the value of T_{slice} for CFQ), whereas more than 16384 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 2.3.2.

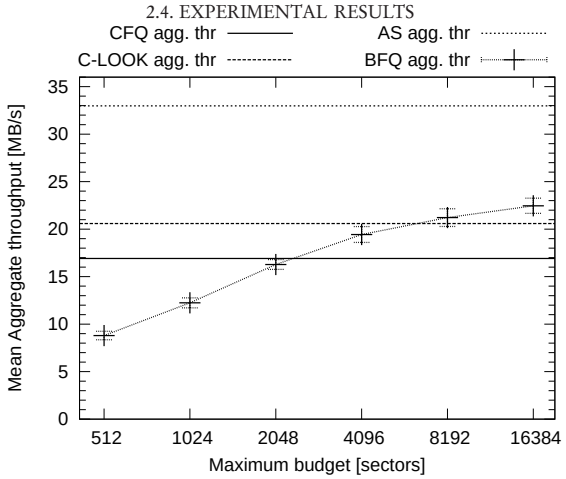


FIGURE 5. Mean aggregate throughput, and associated 95% conf. interval, achieved by BFQ (as a function of B_{max}), CFQ, C-LOOK and AS in case of simultaneous reads of 2, 128 MB long, files.

According to the observations in Section 2.2.3, 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 Figure 5 in case the simultaneous sequential reads of 2, 128 MB long, files are used as worst-case pattern. The aggregate throughput of CFQ, C-LOOK and AS are reported as a reference too. For $B_{max} = 4096$ sectors BFQ guarantees a higher throughput than CFQ.

The second feature we evaluated is the accuracy of the schedulers in providing the desired bandwidth fraction to all the competing applications. For brevity, for BFQ we only report the results for $B_{max} = 4096$ 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% confidence interval for the mean throughput of the i-th application in case of 3 and 4 files was always smaller (or greater) than that obtained in case in case of 5 (or 2) files. This *inclusion property* held also in case of asymmetric allocation (see below). Hence, for brevity, we report only the results for 2 and 5, 128 MB long, files. Finally, we consider only $T_{wait} = 4$ ms for SCAN-EDF and YFQ in the following tables:

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 ($\Delta = 80$ 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 2 files, but it provides a higher throughput than YFQ. Consistently with the arguments in Section 2.3.2, 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 (file read) 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 j-th application is the one with maximum weight, and denoted as S_j the size of the file read by the j-th application, the file read by the i-th 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 because of the incompatibility between the concepts of weight and priority (CFQ), or the lack of any per-application differentiated policy (C-LOOK and AS).

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 2 and 5 files scenarios in the following two tables:

Weight	2	2	2	1	1
Throughput (2 files)	14.62 ±0.60				7.31 ±0.32
Throughput (5 files)	5.60 ±0.08	5.60 ±0.08	5.60 ±0.08	2.81 ±0.04	2.81 ±0.04

Weight	10	2	2	1	1
Throughput (2 files)	26.81 ±0.60				2.77 ±0.16
Throughput (5 files)	16.06 ±0.28	3.25 ±0.07	3.25 ±0.07	1.64 ±0.04	1.65 ±0.04

With regards to SCAN-EDF and YFQ, in accordance with what is said in Sections 2.3.1 and 2.3.2, both failed to guarantee the desired bandwidth distribution in all the experiments. For example, in case of 2 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).

The last two set of experiments were aimed at estimating the per-request completion time guaranteed by the schedulers. In the first one we generated the following Web server-like traffic: 100 processes (all with the same weight/priority) continuously read files one after the other. Every 10 files, each process appended a random amount of bytes, from 1 to 16 kB, to a common log file. Each of the files to read might have been, with probability 0.9, a small 16kB (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. Such a scenario allows the performance of the schedulers to be measured for a mainstream application and, in general, in presence of a mix of both sequential (large files) and

a *seeky* traffic (small files). 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.

Scheduler	**Compl. time small files sec**	**Bandw large files** [MB/s]	**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

As can be seen—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 9 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 files. In general, the low aggregate throughput achieved by SCAN-EDF and YFQ may be imputed to the fact that, differently from all the other schedulers (see Section 2.1.1 for BFQ), they do not take any special countermeasure to keep the throughput high against seeky workloads. 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 almost completely random nature of the workload.

To check whether the guarantee (5) 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 be the last request for that file, which implies $Q_i(t_1^-)+A(t_1,a_i^j)=16kB$ (as explained in Section 2.2.2 T_{FIFO} can be neglected, as requests are all synchronous). 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 i-th 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 $\frac{100*(16+(128-8)+128)+(2048+2*128)}{17.13*1024} = 1.64$ seconds. An even more precise bound would be obtained by considering also the impact of the service that BFQ may overcharge to seeky applications to keep a high throughput (Section 2.1.1).

The last set of experiments was aimed at measuring the ability of the schedulers to support a very time-sensitive application. To this purpose, we set up a VLC streaming server [**VLC**], 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 1 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 1% threshold (this value, as well as the 1 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 performance of the six schedulers). To mimic the mixed workload of a general purpose storage system and to increase to workload so that the disk was the only bottleneck, during each experiment we also run 5 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). The following table 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.

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

As can be seen, with $B_{max} = 4096$ 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, we can consider that according to the last table, just before the end of most experiments, in case of $B_{max} = 4096$ sectors, $24+5$ applications with equal weights are competing for the disk, and the mean throughput is 7.56 MB/s. Moreover, $L_{max} = 256$ sectors in the Linux kernel (by default), and in the experiments we found that, if the i-th application is a streaming thread of the video server,

$L_{i,min} \equiv \min_j L_i^j$ =16 sectors and a budget never higher than 256 sectors is assigned to the application. Hence, according to (6), $d_{i,max} \leq \frac{29*(128-8)+(2048+128)}{7.56*1024} = 0.73$ 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 seconds is added by the rest of the system (probably mostly due to the execution of the 29 threads on the CPU).

2.5. Conclusion

In this chapter we dealt with the problem of providing service guarantees while simultaneously achieving a high disk throughput, in presence of both asynchronous and synchronous requests. With regards to the latter issue, according to the literature research schedulers do not take any countermeasure to tackle the problem of the delayed arrivals of synchronous requests.

In this respect, we proposed BFQ, a new disk scheduler that conceals delayed arrivals through some special timestamping rules. Moreover, thanks also to the its budget-by-budget service model and to the internal B-WF^{2}Q+ budget scheduler, BFQ achieves both high throughput and $O(1)$ deviation (with respect to the number of applications) with respect to the ideal service, over any time interval and under any workload. Especially, the dominant term in both the maximum deviation and the maximum delay is proportional to the maximum budget that can be assigned to an application.

CHAPTER 3

Quick Fair Queueing

QOS provisioning is a long-standing problem whose solution is hindered by business and technical issues. The latter relate to the use and scalability of resource reservation protocols and packet schedulers, which are necessary whenever over-provisioning is not an option.

An IntServ approach can provide fine-grained per-flow guarantees, but the scheduler has to deal with a potentially large number of flows *in progress*[1] (up to 10^5 and more, as reported in a recent study [**KMOR05**]). Besides memory costs to keep per-flow state, the time complexity and service guarantees of the scheduling algorithm can be a concern. A DiffServ approach aggregates flows into a few classes with predefined service levels, and schedules the aggregate classes without need for resource reservation signaling. This drastically reduces the space and time complexity, but within each class, per-flow scheduling may still be needed if we want to provide fairness and guarantees to the individual flows.

The above considerations motivate the interest for packet schedulers with low complexity and tight guarantees even in presence of large number of flows. Round Robin schedulers have $O(1)$ time complexity, but they also have an $O(N)$ worst-case deviation with respect to the ideal amount of service that the flow should receive over any given time interval.

More accurate schedulers have been proposed, based on flow grouping and timestamp rounding, which feature $O(1)$ time complexity and near-optimal deviation from the ideal amount of service (i.e., upper bounded by a constant multiple of the maximum packet size). The two best proposals in this class, the scheduler proposed in [**SBZ99**], hereafter called *Group Fair Queueing—GFQ* for brevity, and SI-WF^{2}Q [**Kar06**] use data structures with somewhat high constants hidden in the $O()$ notation. In particular, on each dequeue operation, GFQ needs to iterate on all groups in which flows are partitioned, whereas SI-WF^{2}Q has to do (in parallel with packet transmissions) a number of operations proportional to the length of the packet being transmitted.

Proposed Solution. We present Quick Fair Queueing (QFQ), a new scheduler with $O(1)$ time complexity, implementing an approximated version of WF^{2}Q+ with near-optimal service guarantees similar to GFQ and SI-WF^{2}Q.

QFQ is based on the partitioning of groups of flows into four sets, each represented by a machine word. All bookkeeping to implement scheduling decisions is based on manipulations of these sets. Multiple groups and flows can be moved at once between sets with simple CPU instructions such as AND, OR, XOR and *Find First bit Set* (FFS).

[1] In [**KMOR05**] a flow is denoted as in progress during any time interval in which the inter-arrival time of its packets is lower than 20 seconds.

Contributions. The major improvement of QFQ over previous proposals is on performance: The algorithm has no loops, and the simplicity of the data structures and instructions involved makes it well suited to hardware implementations. Our Linux x86 version runs in approximately 340 instructions *worst case*[2] per packet enqueue/dequeue with 32k flows. To put the number in context (all worst cases are reported): On the same platform, the DRR scheduler (which boils down to two enqueue or dequeue, but has very poor service guarantees) uses 70 instructions; a simple hash-based classifier uses 202 instructions; the HFSC [**SZN00**] scheduler with 1k flows uses 965 instructions (HFSC's cost is $O(\log N)$) and a full packet processing (from input to output) can easily consume 2000–5000 instructions.

Organization of the Chapter. Section 3.1 presents the QFQ algorithm in detail, Section 3.2 evaluates the service guarantees, ands Section 3.3 complements this introduction by discussing related work. Finally, Section 3.4 measures the performance of the algorithm on a real machine, comparing an actual implementation with the production-quality one of DRR present in Linux.

3.1. Quick Fair Queueing

QFQ approximates WF^{2}Q+, providing near-optimal service guarantees (see Section 3.2) but reducing the implementation complexity to $O(1)$ with small constants thanks to three main techniques. The first two are also widely adopted in the literature [**SBZ99**, **Kar06**, **RP06**]: They are *flow grouping*, which partitions flows into a constant number of groups, and *timestamp rounding*, which reduces the cardinality of sorting keys so that an $O(1)$ bucket sort suffices to sort flows within a group. The third technique, *group sets*, is peculiar to QFQ and is key to a dramatic reduction in the cost of the algorithm: Groups are mapped into four sets, each represented by a single machine word. Due to the properties of the sets, finding the group to schedule only requires a Find First bit Set (FFS) CPU instruction, and all the data structure housekeeping (including changing the state of multiple groups at once) only requires a constant number of bitwise operations on these sets, independently from both the number of groups and the packet length.

3.1.1. Flow Grouping. QFQ groups flows into a small, constant number of groups on which to do the scheduling. A flow k is assigned to a group i defined as

$$i = \left\lceil \log_2 \frac{L^k}{\phi^k} \right\rceil, \tag{7}$$

where L^k is the maximum size of packets for flow k.

The quantity $\sigma_i \equiv 2^i$ (bits) is called the *slot size* of the group. It is easy to see that $L^k/\phi^k < \sigma_i$, hence from (2), $F^k - S^k \leq \sigma_i$ for any flow k in group i.

Because of the definition of groups, for any practical set of L^k's and ϕ^k's in a system, the number of distinct groups is less than 64 (in fact, even 32 groups are largely sufficient in many cases). This is trivially proven substituting values in the equation: As an example, consider that L^k between 64 bytes

[2]We report costs in terms of instruction counts because execution times vary by over one order of magnitude due to cache effects, see Section 3.4. Also, we use the (observed) worst-case counts because even instruction counts vary widely depending on the traffic patterns, and average or best cases would be misleading.

and 16 Kbytes, ϕ_k between 1 and 10^{-6} yield values between $64 = 2^6$ and $16 \cdot 10^9 \approx 2^{34}$, or 29 groups. This small number of groups lets us represent a set of groups with a bitmap that fits in a single machine word.

3.1.2. Timestamp Rounding. QFQ computes S^k and F^k for each flow using the exact algorithm in (2). However, when selecting flows for eligibility or scheduling the next flows to serve, QFQ uses the approximate values[3]:

$$\begin{aligned} \hat{S}^k &\leftarrow \left\lfloor \frac{S^k}{\sigma_i} \right\rfloor \sigma_i, \\ \hat{F}^k &\leftarrow \hat{S}^k + 2\sigma_i. \end{aligned} \tag{8}$$

Furthermore, QFQ replaces $\min_{k \in B(t)} S^k$ with $\min_{k \in B(t)} \hat{S}^k$ in computing (3).

A variant of the *Globally Bounded Timestamp* (GBT) property, proved in Theorem B.1 and Theorem B.2, establishes bounds for $V(t) - \hat{S}^k(t)$ and $\hat{F}^k(t) - V(t)$ that we use to simplify eligibility computation and flow sorting. In particular, at any time $\hat{S}^k$ can never exceed $V(t)$ by more than σ_i, and the range of values for $\hat{S}^k$ is limited to $2 + \lceil L/\sigma_i \rceil$ times σ_i. The limited range and the rounding to multiples of σ_i, implies that the $\hat{S}^k$ can only assume a constant number of different values. Hence, we can sort flows within a group using a constant-time bucket sort algorithm. The use of $\hat{S}^k$ in (3) also saves another sorting step, because, as we will see, the *group sets* defined in the next section will let us compute (3) in constant time.

Once flows in a group are sorted, we can easily compute

$$\begin{aligned} S_i &= \min_{k \in group_i} \hat{S}^k, \\ F_i &= S_i + 2\sigma_i, \end{aligned} \tag{9}$$

which are called the *group's virtual start and finish times*.

The data structure used to sort flows within a group is a *bucket list*—a short array with as many buckets as the number of distinct values for $\hat{S}^k$ (see Figure 1). Each bucket contains a FIFO list of all the flows with the same $\hat{S}^k$ and $\hat{F}^k$. The number of buckets depends on the ratio L/σ_i which is at most $\max(L/L_k)$. For practical purposes, 64 buckets are largely sufficient, so we can map each bucket to a bit in a machine word, and can use a constant-time Find First bit Set (FFS) instruction to locate the first non-empty bucket, which we need to find to select the next flow to serve.

3.1.3. Group Sets and Their Properties. QFQ partitions backlogged groups into four distinct sets, which reduce scheduling and bookkeeping operations to simple set manipulations. Given the small number of groups, each set can be represented with a single machine word, and set manipulations can be implemented with basic CPU instructions such as AND, OR and FFS.

The sets are called **ER**, **EB**, **IR**, **IB** (from the initials of *Eligible, Ineligible, Ready, Blocked*), and the partitioning is done using two criteria:

- **Eligibility:** Group i is said *Eligible* at time t iff $S_i \leq V(t)$, and *Ineligible* otherwise.

[3]There is only one special case where the S^k for certain newly backlogged groups is pushed down to preserve the ordering properties of the **EB** set, described in detail in Section B.2. There it is also proven that this is done without violating service guarantees.

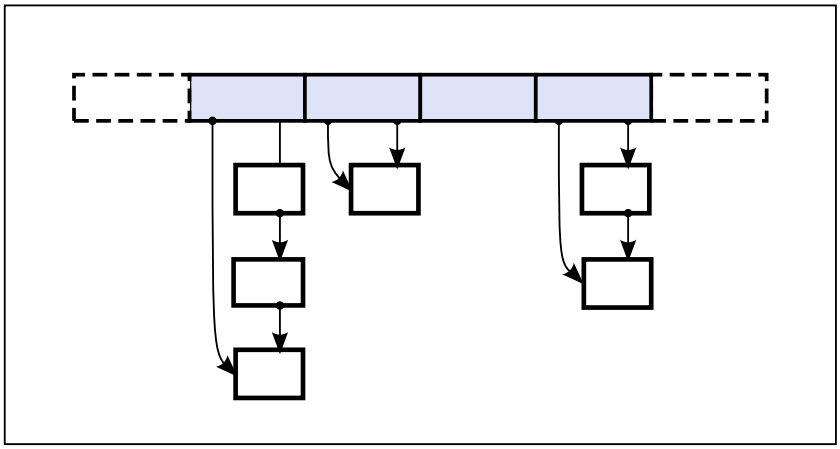


FIGURE 1. A representation of bucket lists. The number of buckets (grey), i.e., possible values of S^k, is fixed and independent of the number of flows in the group. Each list member corresponds to a flow, and the only list operations required are tail insertion and head removal.

- **Readiness:** Independent of its own eligibility, a group i is said to be *Ready* if there is no eligible group $j > i$ with $F_j < F_i$. Otherwise, such a group j exists, and we say that group i is *Blocked* by group j.

Readiness is not intuitive but together with eligibility gives the sets a number of interesting properties, fully exploited in QFQ. In particular, as will be demonstrated later:

- Groups within each set are ordered by group finish times ($\forall j, i \in \mathbf{X} : j > i \implies F_j > F_i$).
- The next packet to serve will always be the head packet of the first flow of the first group in **ER**.
- On dequeue and on virtual time changes, we can move multiple groups at once from one set to another. The groups to be moved are located in an easy-to-find portion of the source set, as explained in Section 3.1.4.

Individually, a group can enter any of the sets when it becomes backlogged or after it is served. Multi-group moves from one set to the other can occur only on the paths

$$\mathbf{IB} \to \mathbf{IR}\ ,\ \mathbf{IR} \to \mathbf{ER}\ ,\ \mathbf{IB} \to \mathbf{EB}\ ,\ \mathbf{EB} \to \mathbf{ER},$$

because transitions in eligibility (driven by changes in $V(t)$) and readiness (driven by changes in the F_i of the blocking group) are not reversible until the group is served.

The ordering properties of sets are proven as follows, and hold over any event that changes set membership:

(1) $\mathbf{IB} \cup \mathbf{IR}$ is sorted by S_i as a result of the GBT property. In fact, if a group i is ineligible, any flow k in the group has $V(t) < S^k < V(t) + \sigma_i$. Due to the rounding we can only have $S_i = \lceil V(t)/\sigma_i \rceil \sigma_i$, and if $i < j$, we have $2\sigma_i \leq \sigma_j$ hence $S_i \leq S_j$.
(2) $\mathbf{IB} \cup \mathbf{IR}$ is also sorted by F_i because of the sorting by S_i and the fact that σ_i are increasing with i.

(3) The sorting of **ER** by F_i is proven in Theorem B.4.

(4) The sorting of **EB** by F_i is proven in Theorem B.5.

By definition, if $i \in$ **EB**, then at least one group $j \in$ **ER** has $F_j \leq F_i$. Because **ER** is sorted by F_i, the readiness test for group i needs only to look at the lowest-order group in **ER** with an index $x > i$. Function `compute_group_state()` in Figure 2 does the computation of set membership for a group.

```
1  // Compute the group's state, see Section 3.1.3
2  compute_group_state(in: group) : state
3  {
4    state = 0 ; // default: not ELIGIBLE, not READY
5    // Find lowest order group x > group.index. This is the one
6    // that may block us.
7    x = ffs_from(set[ER], group.index);
8    if (x == NO_GROUP || groups[x].F >= group.F)
9        state |= R; //READY
10   if (group.S <= V)
11       state |= E; //ELIGIBLE
12   return state;
13 }
```

FIGURE 2. The `compute_group_state()` function implements both the eligibility and the readiness tests and returns the state of the group passed as argument. `ffs_from(d,i)` return the index of the first bit set in d after position i.

3.1.4. Quick Fair Queueing: The Algorithms. We are now ready to describe the full algorithm for the `enqueue()` and `dequeue()` functions, which are executed respectively when a new packet arrives and when the link becomes idle.

Packet Enqueue. The full enqueue algorithm is shown in Figure 3. First of all the packet is appended to the flow's queue, and nothing else needs to be done if the flow is already backlogged (which in turn means that the link is not idle, being the algorithm work-conserving). Otherwise we update the flow's timestamps (lines 8 and 9) and possibly (lines 27 and 28) the group's timestamps (because the slot size is a power of 2, the `round_down()` function just returns the argument with the i least significant bits cleared). We use a constant time bucket sort (line 32) to order the flow with respect to other flows in the group. In case no group was backlogged, and the newly activated flow is the only flow in the system, we check (lines 21–24) if we have to update $V(t)$ as in (2). In case the flow was idle or it activated with a start timestamp lower than the virtual time we have to update the state of the group (lines 29 and 30). The computation of the new state of the group is done in function `compute_group_state()` in Figure 2. Finally, if the group has changed state, we need to move it from one set to another, done by calling function `move_groups()` in Figure 6. The very last step, if the link was idle, is to call `dequeue()`.

Note that an enqueue involves no movement of other groups between sets. $V(t)$ changes only if all other groups are idle, so there are no eligibility changes. If the flow was already backlogged, no group changes its finish time, so there are no readiness changes. If the group j containing this flow just became backlogged, its start time is at least as large as $V(t)$, hence $F_j > \left\lfloor V(t)/\sigma_j \right\rfloor \sigma_j + \sigma_j$. Any Ready group $i < j$ will have $F_i < \lfloor V(t)/\sigma_i \rfloor \sigma_i + 3\sigma_i$ (one σ_i comes from the upper bound on S^k, the other two come from the definition of $F_i = S_i + 2\sigma_i$). Hence $F_i \leq V(t) + 3\sigma_i$. By definition $j > i \implies \sigma_j \geq 2\sigma_i$,

```
1  // enqueue the input pkt of the input flow
2  enqueue(in: pkt, in: flow)
3  {
4    tail_insert(pkt, flow.queue) ; // always enqueue
5    if (flow.queue.head != pkt)
6        return; // Flow is already backlogged, nothing to do.
7    // Update flow timestamps according to (2)
8    flow.S = max(flow.F, V) ;
9    flow.F = flow.S + pkt.length / flow.weight ;
10   g = flow.group ; // reference to the group
11   if (!bucket_empty(g)) {
12       // group already positioned
13       if (flow.S >= g.S)
14           goto skip_update ;
15       // shift the bucket list back of one position
16       bucket_backshift(g) ;
17       // the group was ineligible
18       set[IR] &= ~(1 << g->index) ;
19       set[IB] &= ~(1 << g->index) ;
20   } else {
21       // If there is any backlogged group, at least one is in ER;
22       // otherwise, make sure V ≥ g.S to remain work conserving.
23       if (set[ER] == 0 && V < g.S)
24           V = g.S ;
25   }
26   // compute group timestamps according to (8)
27   g.S = round_down(flow.S, g.slot_size) ; // clear the low i bits
28   g.F = group.S + 2 * g.slot_size ;
29   state = compute_group_state(g) ;
30   set[state] |= 1 << g->index ;
31 skip_update:
32   bucket_insert(flow, g) ; // constant time
33   if (link.idle)
34       dequeue() ;
35 }
```

FIGURE 3. The `enqueue()` function, called on packet arrivals. In the listing, IB, IR, EB, ER are the constants 0..3 corresponding to combinations of the Eligible and Blocked flags. IDLE=4 represents a non-backlogged group.

so $F_j \geq F_i$ and the newly backlogged group j cannot block a previously Ready group, even in the worst case (largest F_i, smallest F_j).

Packet Dequeue. Function `dequeue()` in Figure 4 is called whenever the link is idle and we have packets to transmit. The function returns the next packet to transmit, and updates data structures as needed.

As discussed, the packet selection is straightforward: Upon the call to `dequeue()` at least one flow is eligible, so **ER** is not empty, and we just need to pick (lines 5–7) the group with the lowest index in **ER** and from that the first packet from the first flow. The flow's timestamps are updated, and the flow is possibly reinserted in the bucketlist (lines 10–14).

Next (lines 17–19), the group's timestamps are updated, resulting also in an update of the group's state. If needed, we move the group from **ER** to its new set. If the group has increased its finish time or

```
1  dequeue() : packet //return the next packet to serve
2  {
3    // dequeue the first packet of the first flow of the group in ER
4    // with the smallest index.
5    g = groups[ffs(set[ER])] ;
6    flow = bucket_head_remove(g.bucketlist) ;
7    pkt = head_remove(flow.queue) ;
8    old_V = V ; // Save the old value for the call to make_eligible()
9    V += pkt.length ; // Account for the packet just served
10   flow.S = flow.F ;
11   if (flow.queue.head != NULL) {
12       // Update flow timestamps according to (2)
13       flow.F = flow.S + flow.queue.head.length / flow.weight ;
14       bucket_insert(flow, g) ;
15   }
16   old_F = g.F ; // save for later use
17   if (g.bucketlist.headflow != NULL) {
18       g.S = g.bucketlist.headflow.S ;
19       g.F = g.bucketlist.headflow.F ;
20       if (g.F == old_F)
21           goto skip_unblock ;
22       state = compute_group_state(g) ;
23       set[state] |= 1 << g->index ;
24   } else
25       state = IDLE ;
26   // remove the group from ER if needed
27   if (state != ER)
28       set[ER] &= ~(1 << g->index) ;
29   // if g becomes IDLE, or F has grown, may need to unblock other groups
30   unblock_groups(g.index, old_F) ;
31 skip_unblock:
32   x = set[IR] | set[IB] ;
33   if (x != 0) { // Someone is ineligible, may need to
34                 // bump V up according to (3)
35       if (set[ER] == 0)
36           V = max(V, groups[ffs(x)].S) ;
37       // move from IR/IB to ER/EB all the now eligible groups
38       make_eligible(old_V, V) ;
39   }
40   return pkt ;
41 }
```

FIGURE 4. The `dequeue()` function. See Section 3.1.4 for a full description.

if it has become idle we may need to unblock some other groups (line 30), calling `unblock_groups()`, which is described in the next section.

Finally, since we updated $V(t)$ to account for the newly transmitted packet, we make sure that at least one backlogged group is eligible by bumping up V if necessary, and we move groups between sets using function `make_eligible()`.

Support Functions. We are left with a small set of functions to document, shown in Figure 6, and mostly used in the `dequeue()` code. Function `move_groups()` is trivial and just moves from set `src` to set `dest` those groups whose indexes are included in `mask`. Simple bit operations do the job. Function `make_eligible()` in Figure 6 determines which groups become eligible as $V(t)$ grows after serving a flow. Here again we exploit the features of timestamp rounding to make the operation simple. Figure 5

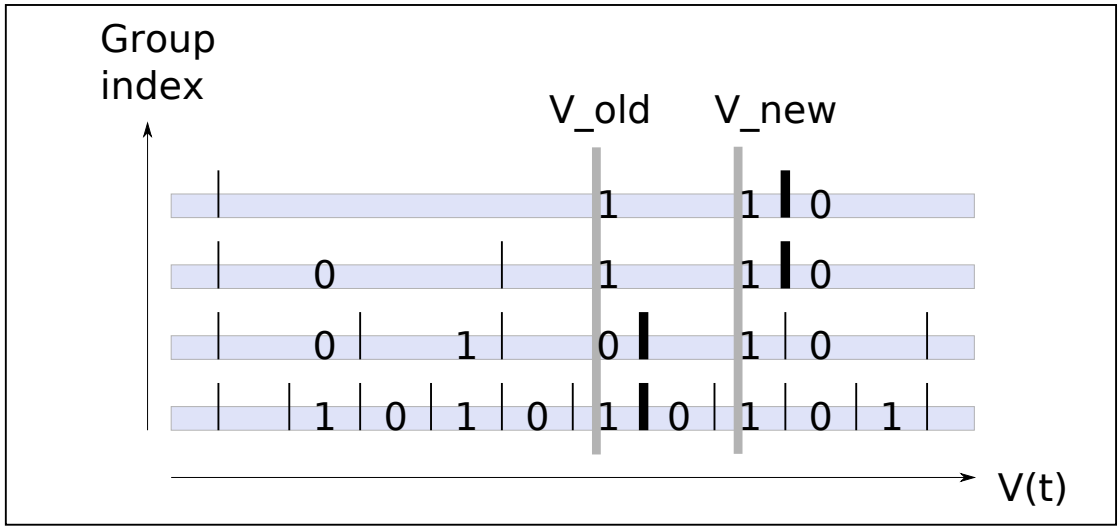


FIGURE 5. How to compute changes in eligibility when $V(t)$ grows. Given $V(t) = V_{old}$, the admissible S_i's for ineligible groups can only assume the discrete values on the tick markers (spaced by $\sigma_i = 2^i$). If we label the intervals on each line as alternating 0's and 1's, the binary representation of $V(t)$ coincides with the sequence of labels of the intervals containing $V(t)$. When $V(t)$ moves from V_{old} to V_{new}, group i becomes eligible iff some bit $j \geq i$ changes in the transition.

gives a graphical representation of the possible values of S_i's and $V(t)$, and the binary representation of $V(t)$. As explained in the caption, if j is the index of the highest bit that changes in the binary representation of $V(t)$, then all backlogged groups with index $i \leq j$ will become eligible. Function `make_eligible()` computes the index j, using an XOR followed by a *Find Last Set* (fls) operation; then computes the set **new_e** that includes all indexes $i \leq j$, and calls function `move_groups` to move groups whose index is in **new_e** from **IR** to **ER** and from **IB** to **EB**.

Function `unblock_groups()` is completely counterintuitive and relies partly on Theorem B.6. Essentially, if, upon serving group i, F_i does not change or coincides with some $F_j, j > i, j \in$ **ER** (line 22–23), all currently blocked group will remain blocked by either group i or group j. In other cases, the theorem shows how we can find the subset of groups (all indexes smaller than i) that must be unblocked.

3.1.5. Time and Space Complexity. From the listings is it clear that QFQ has $O(1)$ time complexity on packet arrivals and departures: All operations, including insertion in the bucket list and finding the minimum timestamps, require constant time. All arithmetic operations can be done using fixed point computations, including the division by the flow weight.

In terms of space, the per-flow overhead is approximately 24 bytes (two timestamps plus one pointer and a few flags), and we have 32..280 bytes per group (to store the slot pointers, two timestamps and some flags).

```
1  // Move the groups in mask from the src to the dst set
2  move_groups(in: mask, in: src, in: dest)
3  {
4      to_move = set[src] & mask ;
5      set[dest] |= to_move ;
6      set[src]  &= ~to_move ;
7  }
8
9  // Move from IR/IB to ER/EB all the groups that become eligible
10 // as V(t) grows from V1 to V2. This uses the logic
11 // described in Figure 5. Please note that,
12 // depending on the representation used for the timestamps an
13 // additional shift could be necessary to correctly place the
14 // mask of the groups to be moved.
15 make_eligible(in: V1, in: V2)
16 {
17     j = fls(V1 XOR V2) ; // highest bit changed in V(t)
18     new_e = (1 << (j+1)) - 1 ;
19     move_groups(new_e, IR, ER) ;
20     move_groups(new_e, IB, EB) ;
21 }
22
23 // Possibly unblock groups after serving group i with F=old_F
24 unblock_groups(in: i, in: old_F)
25 {
26       x = ffs_from(set[ER], i + 1)] ;
27       if (x == NO_GROUP || groups[x].F > old_F) {
28           // Unblock all the lower order groups (Theorem B.6)
29           low_groups = (1 << i) - 1 ;
30           move_groups(low_groups, EB, ER) ;
31           move_groups(low_groups, IB, IR) ;
32       }
33 }
```

FIGURE 6. Support functions to recompute the set of eligible groups after a flow has been served. These are described in Section 3.1.4

.

3.2. Service Properties

We report here both the B-WFI (bit guarantees) and the T-WFI (time guarantees) of QFQ.

3.2.1. Bit guarantees. The B-WFI^k guaranteed by a scheduler to a flow k is defined as:

$$\text{B-WFI}^k \equiv \max_{[t_1,t_2]} \phi^k W(t_1, t_2) - W^k(t_1, t_2), \tag{10}$$

where $[t_1, t_2]$ is any time interval during which the flow is continuously backlogged, $\phi^k W(t_1, t_2)$ is the minimum amount of service the flow should have received according to its share of the link bandwidth and $W^k(t_1, t_2)$ is the actual amount of service provided by the scheduler to the flow. This definition is indeed slightly more general than the original one, where t_2 is constrained to the completion time of a packet.

THEOREM 3.1 (B-WFI for QFQ). *For a flow k belonging to group i QFQ guarantees*

$$\text{B-WFI}^k = 3\phi^k \sigma_i + 2\phi^k L. \tag{11}$$

The proof does not differ in principle from the proof of the B-WFI for WF^{2}Q+ [**BZ97**], and can be found in Section B.3. As a term of comparison, in a perfectly fair ideal fluid system like the GPS server B-WFI$^k = 0$ [**BZ97**], whereas repeating the same passages of the proof in case of exact timestamps, i.e., exact WF^{2}Q+ *with stepwise* $V(t)$, the resulting B-WFI^k would be $(L^k + 2\phi^k)L$.

3.2.2. Time Guarantees. Expressing the service guarantees in terms of time is only possible if the link rate is known. The T-WFI^k guaranteed by a scheduler to a flow k on a link with constant rate R is defined as:

$$\text{T-WFI}^k \equiv \max\left(t_c - t_a - \frac{Q^k(t_a^+)}{\phi^k R}\right), \tag{12}$$

where t_a and t_c are, respectively, the arrival and completion time of a packet, and $Q^k(t)$ is the backlog of flow k.

THEOREM 3.2 (T-WFI for QFQ). *For a flow k belonging to group i QFQ guarantees*

$$\text{T-WFI}^k = \left(3\left\lceil\frac{L^k}{\phi^k}\right\rceil + 2L\right)\frac{1}{R}. \tag{13}$$

For the proof, once again see Section B.3. For comparison, a perfectly fair ideal fluid system would have T-WFI$^k = 0$, whereas for WF^{2}Q+, which uses exact timestamps, repeating the same passages of the proof yields T-WFI$^k = (\frac{L^k}{\phi^k} + 2L)/R$.

3.3. Related Work

Round Robin (RR) schedulers lend naturally to $O(1)$ implementations with small constants. Several variants have been proposed, as e.g., Deficit Round Robin [**SV95**], Smoothed Round Robin [**Chu01**], Aliquem [**LMS04**] and Stratified Round Robin [**RP06**], to address some of the shortcomings (burstiness, etc.) of RR schedulers. As we have seen in Section 1, one unfixable feature of this family of schedulers is that, irrespective of the flow's weight ϕ^k, their T-WFI has an $O(NL)$ component[4], where L is the maximum packet size and N is the total number of flows in the system.

Timestamp-based schedulers are by their own nature constrained to serve flows in some timestamp order, so they pay the cost of keeping flows ordered, which has $\Omega(\log N)$ cost. As we have seen in Chapter 1, schedulers such as WF^{2}Q [**BZ96**] and WF^{2}Q+ [**BZ97**] offer *optimal* WFI, i.e., the lowest possible WFI for a non-preemptive system, but have an $O(\log N)$ time complexity. Approximated variants of these schedulers use rounded timestamps instead of exact ones to get rid of the burden of exact ordering and run in $O(1)$ time. Examples are GFQ [**SBZ99**], SI-WF^{2}Q [**Kar06**] and the QFQ algorithm described here.

The approximation has an implication: As proved in [**XL02**], in any system using approximate timestamps, the difference between the packet completion times in the real system and in an ideal GPS system is larger than $O(1)$. However, even optimal schedulers such as WF^{2}Q and WF^{2}Q+ where this difference is $O(1)$, feature T-WFIs of the same order, that we find in the approximated schedulers just

[4]In WF^{2}Q and WF^{2}Q+ this component is $O(L(1 + 1/\phi^k))$. This can still grow to $O(NL)$ for low-weight flows, but high weight flows receive better treatment.

mentioned (GFQ, SI-WF^{2}Q, QFQ), which is $O(L(1+1/\phi^k))$. The difference is only in the multiplying constant, which is 1 with exact timestamps and slightly larger otherwise (e.g., 3 in the case of QFQ—see Sec.3.2.2). Thus, approximated timestamps still give much better guarantees than Round Robin schedulers. In addition, the optimal T-WFI itself necessarily contains a component equal to one time the worst-case packet service time as well, while the T-WFI of these approximate schedulers differs from the optimal one in that it contains a component equal to a few times the worst-case packet service time. In this sense, these schedulers do achieve a *near-optimal* T-WFI. Using the same approach adopted to prove the B-WFI of QFQ, it would be easy to show that they also achieve near-optimal B-WFI.

Approximated timestamp-based schedulers typically use data structures that are more complex to manipulate than those used in Round Robin or exact timestamp-based schedulers. As a consequence, the same or better asymptotic complexity does not necessarily reflect in faster execution times. As an example, the approximated variants of WF^{2}Q+ presented in GFQ [**SBZ99**] use timestamp rounding, flow grouping and a calendar queue to maintain a partial ordering among flows within the same group. Each scheduling decision requires a linear scan of the groups in the system to determine candidate for the next transmission. The actual algorithms are only outlined, and no public implementation is available; the authors claim a sustainable rate of 1 Mpps and a per-packet overhead of 500 ns for their hardware implementations.

LFVC [**SVC97**] rounds timestamps to integer multiples of a fixed constant. Reducing the timestamps to a finite universe enables LFVC to use data structures like van Emde Boas priority queues, which have $O(\log\log N)$ complexity for all the basic operations they support. This is not $O(1)$ but grows very slowly with N, though the van Emde Boas priority queues have high constants hidden in the $O()$ notation. A drawback of LFVC is that its worst-case complexity is $O(N)$, because the algorithm maintains separate queues for eligible and ineligible flows, and individual events may require to move most/all flows from one queue to the other.

Finally, SI-WF^{2}Q is based on special data structures called Interleaved Stratified Timer Wheels (ISTW). ISTW containers have several nice properties that allow SI-WF^{2}Q to execute packet enqueue and dequeue operations at a worst-case cost independent of even the number of groups, provided that a number of groups proportional to the maximum packet size (in the worst case) is moved between two containers during the service of each packet. In a system where the latter task can be performed, and completed, during the transmission of a packet, SI-WF^{2}Q runs at an $O(1)$ amortized cost per packet transmission time.

3.4. Experimental Results

The most interesting feature of QFQ is its bounded (independent of number of flows) and small per-packet execution time, which makes the algorithm extremely practical and amenable to really fast implementations. This section analyzes the performance of our Linux implementation of QFQ[5], and compares it with the Deficit Round Robin (DRR) scheduler available on the same system.

The choice of DRR is relevant because it represents a practical lower bound for the complexity of a scheduler. It would have been interesting to evaluate GFQ or SI-WF^{2}Q as well, but the lack of a full

[5]The source code of the scheduler and all the other software used to obtain the results presented in this section are available, together with all the data collected during the measurements, at [**qfq**].

specification and public implementation of these algorithms prevented us from doing so. Implementing these schedulers from scratch would have required an exceeding amount of coding, well outside the scope of this work; and, especially, implementation decisions not fully covered by the respective papers could have made the results not representative of the original intentions of their authors, leading to an unfair comparison.

The experiments have been ran using three PCs acting respectively as source, router, and sink of network traffic. Source and router were connected by a 1 Gbit/s card, whereas router and sink were connected by a 100 Mbit/s card. The source generated UDP traffic using a modified version of the in-kernel traffic generator, `pktgen` [**Ols05**], with different configurations. Here we focus on two of them:

- **Light load:** Packet length is uniformly distributed between 64 and 1518 bytes, a 100 Mbit/s link connects the source host and the router.
- **Heavy load:** Packets are 64 bytes long and a 1 Gbit/s link connects source and router.

The two scenarios serve to exercise different code paths in the schedulers, as the execution times may depend on the backlog status of flows, groups and sets as well as on the content of the cache and the interleaving with other system activities). In particular, low load is a worst-case scenario (instruction-wise) for QFQ, because there is a lot of bookkeeping involved in every `enqueue()` and `dequeue()`. Conversely, at high load, QFQ's `enqueue()` is as inexpensive as in DRR, and `dequeue()` is normally cheap because the current group is often backlogged.

When doing measurements on a machine with deep memory hierarchies (our CPU was a 3.0 GHz, dual core, Intel E8400) and code with variable execution paths, single performance numbers are insufficient to describe the behavior of the system. As a consequence the evaluation has been done by measuring three parameters across each execution of the `enqueue()` and `dequeue()` functions: The number of *retired instructions*, the total *CPU cycles*, and the number of *cache misses*. All of them were computed by reading the relevant CPU performance counters around the functions being measured. Samples were stored in a kernel ring buffer of 64k elements, and dumped to userspace every 5 seconds. To let the system settle, no dumps were done in the first 15 seconds of the experiment, and the first 10 dumps were ignored. We used the next 30 dumps for our evaluation. We took care of preventing major sources of imprecision, such as interrupts and preemptions/migrations to interfere with the measurements, by disabling interrupts around the measured sections of code, and pinning the relevant interrupt handlers to a single core whenever possible.

Given the potentially high variance of the samples, before computing any statistics it is important to look at the distribution of the data. Figure 7 shows the distributions of the instruction counts for enqueue/dequeue in various configurations, accumulated across multiple experiments. We only show a total of 50k samples in these graphs, but distributions are essentially unmodified as the number of samples increases. As expected, instruction counts proved to be stable across multiple executions. and they flat regions of the curves correspond to the different code paths in the functions under test.

DRR is mostly oblivious to the state of the system. Its `enqueue()` function just appends the packet to the flow's queue, and appends the flow to the FIFO list if not there yet, so the instruction count is practically flat. DRR's `dequeue()` also has a mostly constant instruction count irrespective of the load.

QFQ's `enqueue()` also starts by enqueueing the packet, but then possibly has to do some extra work depending on the flow/group/system state. The frequency of extra work is higher at low load, when flows and groups are almost always idle—in fact, the shortest code path is never exercised in our low-load scenario, and the instruction count for QFQ's `enqueue()` varies between 180 and 240. At high load, the function has an instruction count similar to DRR's `enqueue()` (55 instructions) in 80–90% of the cases, while the remaining ones exercise different code paths, up to the same 240 instructions as before.

QFQ's `dequeue()` is influenced by the load in a similar way. At low load the cost is always around 155 instructions, because in this scenario there is always only one backlogged flow. At high load the instruction count varies, again due to the existence of multiple code paths which are taken on different executions. We range from a minimum of 130 instructions to a maximum of 340 instructions (only in 2–3% of the cases; the 95-percentile value is around 270 instructions).

Note that the worst case for QFQ's `dequeue()` is the same with 1k and 32k flows—the curves have different shapes only because the relative frequency of the various code paths changes.

Moving from instruction to cycle counts the nice and flat curves we have seen so far disappear. Most of this depends on the effect of caches and of instruction latencies/dependencies. A small part also depends on measurement errors caused by the effect of hardware factors like pipelining and out-of-order execution on short observation periods.

The effects are visible on all configurations. Due to space limitations we focus on one set of curves, namely the distribution of durations, in CPU cycles, of the `dequeue()` operation under high-load conditions. The four curves (for QFQ and DRR, 1k and 32k flows) are shown in Figure 8. With low load, and in the best runs, a warm cache allows the delivery of multiple instructions per clock cycle, but apart from some lucky runs, especially as the number of flows grows, the cache becomes less and less effective, and execution times quickly ramp up for both QFQ and DRR. The two algorithms have $O(1)$ cost, but here the curves with 1k and 32k flows are different for both algorithms. A large number of flows implies a larger memory footprint and a higher chance of cache misses. Even though misses are limited (less than 2–3 in most cases), they are so expensive that the execution time of the functions can be affected badly—near the end of the distribution, both DRR and QFQ have some executions that consume over 2000 clock cycles, compared to the best values of 54 and 117, respectively. Such a large deviation explains why performance claims should be taken with extreme care by the reader, if not backed up by actual experimental data. Moreover, given the nature of the architecture in consideration, the real worst-case execution times can hardly be estimated.

With these considerations in mind, the following table summarizes the the average per-packet execution times, and the 95% confidence intervals, measured for QFQ and DRR in the worst performing scenario (duration-wise), which proved to be the heavy load. The purpose of the table is not to provide single numbers to characterise the two scheduler, but rather to warn the reader once more on the care that must be used in analysing these measurements. As an example, note how, even though the instruction counts are relatively repeatable, some of the entries have a large variance which reflects the multi-modal results of the measurements.

Moving to execution times (our machine has a fixed 3 GHz clock speed), as expected we see that the variance explodes, and the ratio between instructions and times changes widely from one configuration to another.

	1k Flows		32k Flows	
	QFQ	DRR	QFQ	DRR
Instructions				
enqueue	70.1 ± 42.2	51.1 ± 0.8	56.9 ± 10.6	51 ± 0
dequeue	153.9 ± 51.0	68.2 ± 2.1	158.5 ± 55.3	69.2 ± 5.7
Duration, in ns				
enqueue	40.2 ± 46.9	40.9 ± 45.1	62.1 ± 63.8	56.5 ± 61.8
dequeue	117.0 ± 65.0	25.9 ± 22.2	212.6 ± 105.3	39.1 ± 67.0

TABLE 1. Per-packet instructions/execution times.

3.5. Conclusion

In this chapter we presented QFQ, an approximated implementation of WF^{2}Q+ which can run in constant time, with very low constants and using extremely simple data structures. On each packet arrival/departure, the algorithm executes only one multiplication and a small number of arithmetic and logic operations. The simplicity of QFQ is witnessed by its low instruction count (340 per packet worst-case) and fast execution times.

In addition to a detailed description of the algorithm and a thorough theoretical analysis of the service guarantees, we also provided a detailed evaluation of the performance on real hardware, both in terms of instructions and CPU cycles. We have also developed a production quality implementation of QFQ which is, to the best of our knowledge, the first publicly available implementation of a scheduler of this class ($O(1)$ time and near-optimal WFI). The code, which works with the Linux Packet Scheduling framework, has been used for our measurements and is available at [**qfq**].

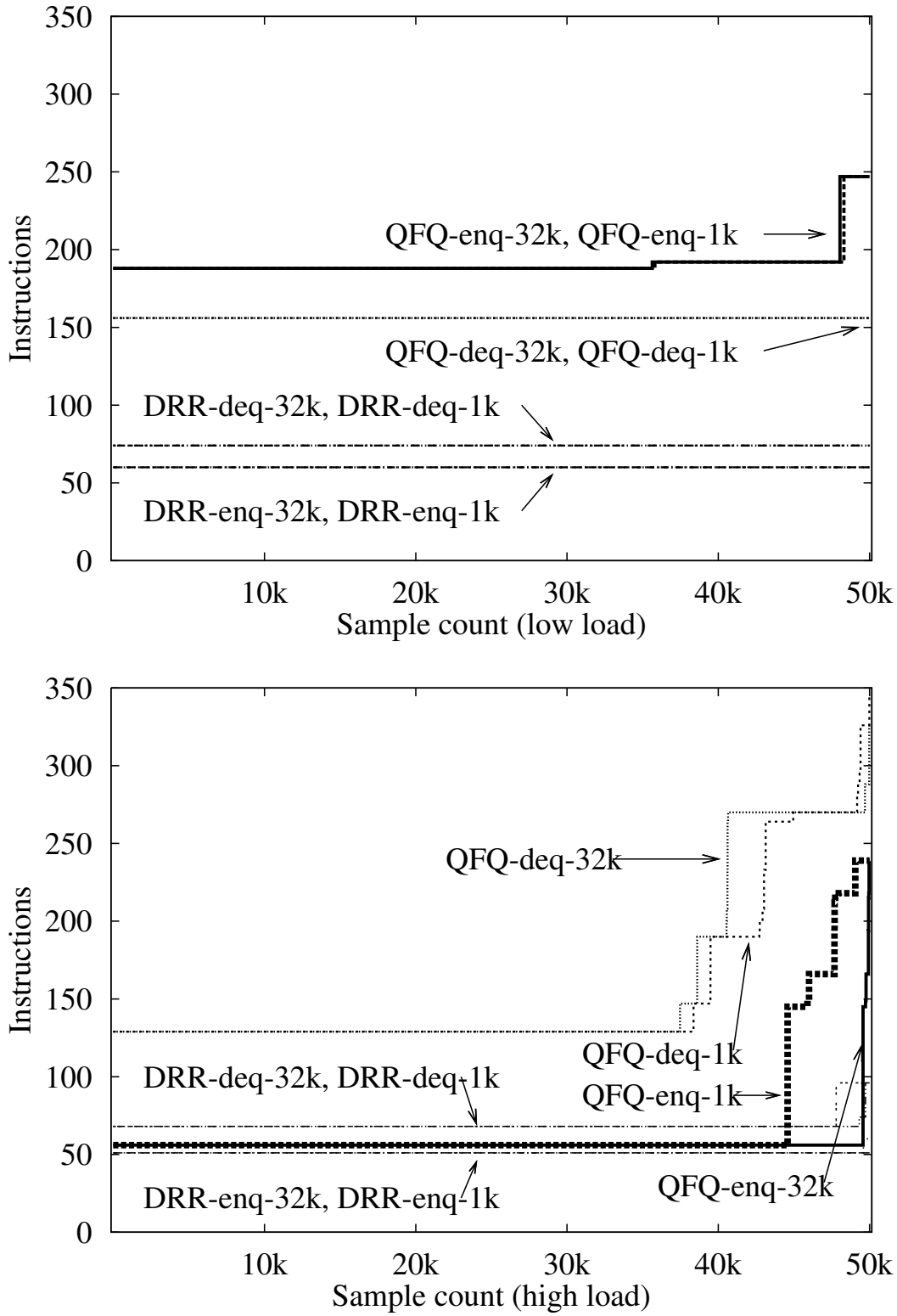


FIGURE 7. Distribution of the number of instructions.

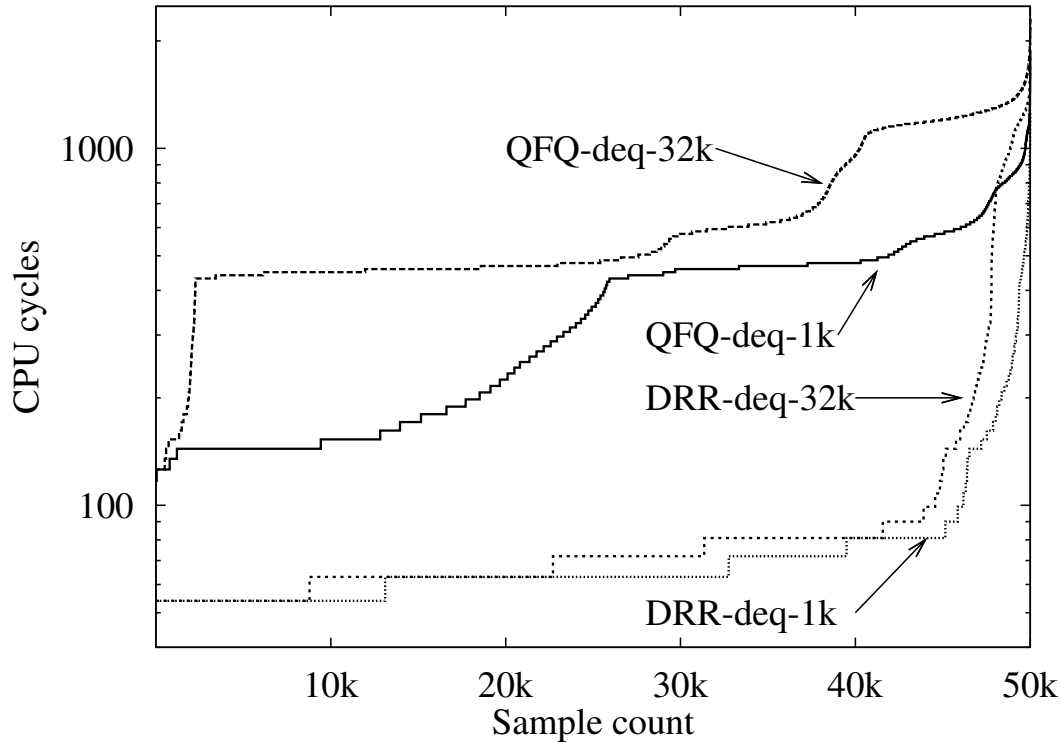


FIGURE 8. Distribution of CPU cycles, dequeue.

APPENDIX A

Proofs of BFQ Service Properties

BFQ properties were described in Chapter 2 by Theorem 2.1 and Theorem 2.2; in this section we will prove them, but, before proceeding, we first need to introduce some additional notations and prove a lemma used in both proofs.

Since it is often used, for brevity we call *total service property* the fact that, at any time t, the total amount of service $W(t)$ provided by the real and the ideal system during $[0, t]$ is the same. Moreover, since we focus only on what happens inside B-WF^{2}Q+, for simplicity from now on we neglect the issues related to deceptive idleness. Especially, since the state of B-WF^{2}Q+ changes only in consequence of request arrivals and service, without losing generality, we assume that idling periods have zero duration, and we say that an application is backlogged iff its request queue inside B-WF^{2}Q+ is not empty.

Given a budget $B = B_i^j$ assigned to an application, we denote as $R(B)$ the batch served using B. If an application becomes idle before consuming all of its budget B, or if it is deactivated because the remaining budget is lower than the size of the next request to serve, then the size of the batch $R(B)$ is lower than B. In this case we say that the budget is *underutilized*.

Suppose that only a portion $B' < B$ of an underutilized budget is used by an application. According to the code in Figure 4, three points can be highlighted. First, the virtual finish time of the batch is higher than it would have been if computed as a function of B'. Hence the start of the service of $R(B)$ may be *delayed*. Second, this possibly delayed start, plus the fact that the disk is released *in advance*, are only beneficial for the completion time of the requests of the other applications. Third, when the application is done using B, its virtual finish time is properly decreased, so as to account only for the actual service received. Hence the next virtual start time will be correctly computed. In the end, the first issue is the only one to be addressed in computing worst-case guarantees. In this respect, in the proof of Lemma A.2, we start from a fictitious scenario where a longer batch $\overline{R}(B)$ is served instead of $R(B)$.

Given any quantity A defined for the real system, we use the superscript notation A^I to denote its counterpart in the ideal system. For example, we denote as $W_i^I(t)$ the amount of service received by the i-th application in the ideal system during $[0, t]$. By definition, $V(t)$ is equal to the *normalized* amount of service that would be guaranteed by the fluid system to a continuously backlogged application during $[0, t]$ with respect to its weight, i.e., $W_i^I(t) \geq \phi_i \cdot V(t)$ holds for any application i continuously backlogged during $[0, t]$. Moreover, we denote as $V_i(t)$ the virtual time of the i-th application. While the application is backlogged, $\frac{dV_i(t)}{dt} = \frac{1}{\phi_i} \cdot \frac{dW_i^I(t)}{dt}$, i.e., the growth of $V_i(t)$ measures the normalized amount of service received by the application in the ideal system. On the contrary, while the application is idle, $V_i(t)$ is fictitiously increased as follows: $V_i(t) \leftarrow \max(V_i(t^-), V(t))$. The ideal system

serves applications in such a way to guarantee that $\forall i, t\ V_i(t) \geq V(t)$. Finally, differently from the real system, it serves the requests of each application in FIFO order.

We denote as $F_i(t)$ the virtual finish time of the last batch, of the i-th application, already completed in the real system at time t. Recall that the real system starts serving a new batch only if it is eligible. Hence, from the time-stamping rules in Figure 4, it follows that

$$F_i(t) \leq V(t) + \frac{B_{i,max}}{\phi_i}. \tag{14}$$

The second of the following lemmas will be used as a building block in proving both theorems.

LEMMA A.1. *Let $R(B^1)$, $R(B^2)$,..., $R(B^k)$ be k batches (possibly of different applications) consecutively served in the real system during a time interval $[t_s, t_f]$ ($R(B^1)$ starts at time t_s, while $R(B^k)$ is completed at time t_f). Suppose that:*

(1) *The related k budgets are fully utilized.*
(2) *The k batches are completed in the real system in the same order as in the ideal system.*
(3) *There is a time instant $t_0 \leq t_s$ such that at time t_0 the ideal system has not yet served any sector of any of the k batches.*

Then, denoted as t_f^I the time instant at which all the k batches are completed in the ideal system, $W(t_f) - W(t_f^I) \leq W(t_0, t_s)$. Especially, if $t_s = 0$, we have that $W(t_f) - W(t_f^I) \leq 0$, and hence $t_f \leq t_f^I$.

PROOF. Since the batches are completed in the same order in both systems, to finish $R(B^k)$, the ideal system must complete all the k batches. In addition, the ideal system might serve other batches during $[t_0, t_f^I]$. Hence, considering also the total service property, $W(t_0, t_f^I) \geq \sum_{i=1}^{k} B^k = W(t_s, t_f) = W(t_0, t_f) - W(t_0, t_s)$. This inequality allows the thesis to be proven as follows:

$$\begin{aligned} W(t_f) - W(t_f^I) = W(t_0, t_f) - W(t_0, t_f^I) \quad \leq \\ W(t_0, t_f) - W(t_0, t_f) + W(t_0, t_s) = W(t_0, t_s). \end{aligned} \tag{15}$$

□

Before the second lemma, we need to define a last function. We define as lag of the i-th application at time t, the quantity

$$\text{lag}_i(t) \equiv W_i^I(t) - W_i(t). \tag{16}$$

LEMMA A.2. *Let R_i^j be the j-th request issued by the i-th application, and let s_i^j and c_i^j be its start and completion time. We have that*

$$\text{lag}_i(s_i^j) \leq B_{max} + L_{max}. \tag{17}$$

Moreover, if $\forall t \in [t_1, c_i^j]\ \frac{dW(t)}{dt} = T_{agg}$ and all the applications are continuously backlogged in the ideal system, then, defined R_i^m as the last requests served in the real system among the ones arrived during $[a_i^j, a_i^j + T_{FIFO})$ (possibly $R_i^m = R_i^j$),

$$c_i^j \leq c_i^{I,m} + \frac{B_i^l - L_i^j}{\phi_i T_{agg}} + \frac{B_{max} + L_{max}}{T_{agg}}, \tag{18}$$

where B_i^l is the budget used to serve R_i^j.

PROOF. For the moment, we assume that $V(t)$ is smoothly tracked, and we postpone to the end of the proof the evaluation of the consequences of the stepwise approximation of $V(t)$ (outlined in Section 2.1.4). We proceed as follows. First we consider a fictitious batch $\overline{R}(B_i^l)$ in which a fake request R_{fake} is added after the last request of the actual batch $R(B_i^l)$. We suppose that R_{fake} is long enough to let the whole budget B_i^l be consumed ($L_{fake} = B - \sum_{R_i^j \in R(B)} L_i^j$). We consider what would have happened if the fictitious batch $\overline{R}(B_i^l)$ would have been served in place of $R(B_i^l)$. We define this scenario as the *fictitious* scenario, as opposed to the *actual* scenario. Then we compute an upper bound to $\text{lag}_i(s_i^j)$ and to c_i^j as a function of quantities computed for the fictitious scenario.

With regards to c_i^j, due to the C-LOOK/FIFO service order of the real system, in addition to the requests queued at time a_i^{j-}, the request R_i^j may also wait for the service of the other requests arrived during $[a_i^j, a_i^j + T_{FIFO})$. In contrast, the ideal system serves the requests of each application in FIFO order. Hence in the real system R_i^j is served as if $c_i^{I,j} = c_i^{I,m}$.

Given the value of a quantity q for the actual scenario, we use the over-line notation $\overline{q}$ to refer to the corresponding value for the fictitious scenario. Hence, $\overline{c}_i^l \geq s_i^j$ and $\overline{c}_i^{I,l} \geq c_i^{I,j}$ are e.g., the finish time of $\overline{R}(B_i^l)$ in the real and in the ideal system for the fictitious scenario. Note that $\forall t \leq c_i^j \ \overline{W}(t) = W(t)$. Finally, let D_i^j be the sum of the sizes of the requests in $R(B_i^l)$ served after R_i^j, plus the size L_i^j of R_i^j itself.

Since we are computing worst-case guarantees, without losing generality, we assume that all the batches served before $\overline{R}B_i^l$ are fully utilized. We consider two cases. First, all the batches served during $[0, \overline{c}_i^l]$ in the real system for the fictitious scenario are completed in the same order in which they are completed in the ideal system. In this case, from Lemma A.1, and considering that $c_i^{I,j} = \overline{c}_i^{I,j}$ and all the applications are continuously backlogged during $[c_i^{I,j}, \overline{c}_i^{I,l}]$, we have that $c_i^j \leq \overline{c}_i^l \leq \overline{c}_i^{I,l} \leq c_i^{I,j} + \frac{B_i^l - L_i^j}{\phi_i T_{agg}}$. Hence (18) holds. Moreover, considering that $s_i^j \leq \overline{c}_i^{I,l}$ it also follows that, for the actual scenario, $W_i^I(s_i^j) \leq W_i(s_i^j) + D_i^j \leq W_i(s_i^j) + B_{max}$. Hence, (17) holds too.

Second, some of the batches served before $\overline{R}(B_i^l)$ in the real system for the fictitious scenario are completed after $R(B_i^l)$ in the ideal system. Let $R(B^m)$ be the last of these batches, and suppose it starts to be served at time t_0 in the real system. Let $R(B^{m+1})$, $R(B^{m+2})$, ..., $R(B^{m+k}) = \overline{R}(B_i^l)$, be the successive batches served in the real system. Since the ordering among virtual finish times is the same as among the completion times in the ideal system of the corresponding batches [**BZ97**], the virtual finish time of these k batches is lower than the virtual finish time of $R(B^m)$. Hence, for B-WF^{2}Q+ to choose $R(B^m)$ instead of one them, all these k batches have to be not yet eligible or not yet arrived at time t_0. In both cases, the ideal system has not provided any service to any of them at time t_0. We also assume that $s_i^j \geq \overline{c}_i^{I,l}$, since, if the thesis holds in this (sub)case, it trivially holds also in case $s_i^j < \overline{c}_i^{I,l}$. Hence, thanks to Lemma A.1, in the fictitious scenario, $\overline{W}_i^I(\overline{s}_i^{I,j}, s_i^j) = \overline{W}_i^I(\overline{s}_i^{I,j}, \overline{c}_i^{I,l}) + \overline{W}_i^I(\overline{c}_i^{I,l}, s_i^j) = D_i^j + \overline{W}_i^I(\overline{c}_i^{I,l}, s_i^j) \leq D_i^j + W(\overline{c}_i^{I,l}, \overline{c}_i^l) - D_i^j \leq D_i^j + B^m - D_i^j = B^m \leq B_{max}$. Besides, since $s_i^j = \overline{s}_i^j$, $W_i^I(s_i^j) \leq \overline{W}_i^I(s_i^j)$ (because also the additional fake request R_{fake} may have been served in the fictitious

scenario), $\overline{W}_i^I(\overline{s}_i^{I,j}) = \overline{W}_i(s_i^j)$, and $\overline{W}_i^I(s_i^j) = \overline{W}_i^I(\overline{s}_i^{I,j}) + \overline{W}_i^I(\overline{s}_i^{I,j}, s_i^j)$, then $W_i^I(s_i^j) \leq W_i(s_i^j) + B_{max}$ in the actual scenario. This proves (17).

To prove (18), suppose for a moment that the real system is able to immediately preempt $R(B^m)$ as any of the batches $R(B^{m+1})$, $R(B^{m+2})$, ..., $R(B_i^l)$ becomes eligible. Thanks to the hypothesis of the applications being continuously backlogged, this ability would not affect the request arrival pattern, and hence the order in which these k batches would be served in the ideal system. Hence $c_i^{I,j}$ would not change. Moreover, thanks to Lemma A.1 and to the arguments used for the previous case, $c_i^j \leq c_i^{I,j} + \frac{B_i^l - L_i^j}{\phi_i T_{agg}}$ would hold. Unfortunately, due to the impossibility of preempting $R(B^m)$, the real system starts serving the j batches with a delay of at most $\frac{B_{max}}{T_{agg}}$ time units, hence $c_i^j \leq c_i^{I,j} + \frac{B_i^l - L_i^j}{\phi_i T_{agg}} + \frac{B_{max}}{T_{agg}}$.

Finally, for the reasons explained in Section 2.2.1, the stepwise approximation of $V(t)$ causes the additional worst-case L_{max} and $\frac{L_{max}}{T_{agg}}$ components, respectively.

□

A.1. Service Guarantees

To prove the theorem, we first compute the maximum per-application deviation (lead/lag) of the real system with respect to the ideal system.

LEMMA A.3. *At any time instant t, and for all $i \in \{1, 2, \ldots, N\}$, the following inequalities hold:*

$$\min\left(\phi_i(V_i(t) - F_i(t)), 0\right) \leq lag_i(t) \leq B_{max} + L_{max}. \tag{19}$$

PROOF. Once granted access to the disk, and until budget exhaustion, an application receives an amount of service equal to the total amount of service provided by the system. Hence, thanks to the total service property, $\text{lag}_i(t)$ cannot decrease while the i-th application is being served. This has two consequences.

First, $\exists s_i^l : \forall t\ \text{lag}_i(t) \leq \text{lag}_i(s_i^l)$, where s_i^l is the start time of a batch $R(B_i^l)$. Hence the rightmost inequality in (19) follows from Lemma A.2. Second, $\exists c_i^l : \forall t\ \text{lag}_i(t) \geq \text{lag}_i(c_i^l)$, where c_i^l is the completion time of a batch $R(B_i^l)$. If $F_i(c_i^l) \leq V_i(c_i^l)$, then $\text{lag}_i(c_i^l) \geq 0$ and the leftmost inequality in (19) holds.

To prove the thesis in case $F_i(c_i^l) > V_i(c_i^l)$, let s_i^l and $s_i^{I,l}$ be, respectively, the time instants at which the budget starts to be served in the real and in the ideal system. We have that $W_i^I(s_i^{I,l}) = W_i(s_i^l)$. It follows that

$$W_i^I(c_i^l) - W_i(c_i^l) = W_i^I(s_i^{I,l}, c_i^l) - W_i(s_i^l, c_i^l). \tag{20}$$

Since $F_i(c_i^l) > V_i(c_i^l)$, the ideal system is still serving the budget $R(B_i^l)$ at time c_i^l, which implies $W_i^I(s_i^{I,l}, c_i^l) = \phi_i\left(V_i(c_i^l) - S_i^l\right)$, where S_i^l is the virtual start time of the batch $R(B_i^l)$. On the other hand, $W_i(s_i^l, c_i^l) \leq \phi_i\left(F_i(c_i^l) - S_i^l\right)$. Substituting this inequality and the previous equality in (20), we get $W_i^I(c_i^l) - W_i(c_i^l) \geq \phi_i\left(V_i(c_i^l) - S_i^l\right) - \phi_i\left(F_i(c_i^l) - S_i^l\right) = \phi_i\left(V_i(c_i^l) - F_i(c_i^l)\right)$, which proves the thesis. □

We can now restate and prove Theorem 2.1.

THEOREM 2.1. *For any time interval $[t_1, t_2]$ during which the i-th application is continuously quasi-backlogged, BFQ guarantees that:*

$$\phi_i \cdot W(t_1, t_2) - W_i(t_1, t_2) \leq B_{max} + B_{i,max} + L_{max}. \tag{21}$$

PROOF. Let a_i^j be the arrival time of the request that is waiting to be completed at time t_1. From the timestamping rule in Figure 4, line 8, and from the fact that the i-th application is continuously backlogged, it follows that the virtual start time of its batches served during $[t_1, t_2]$ is the same as if all these batches arrived back-to-back. Moreover, the service provided by the real system to the i-th application does not depend on the arrival pattern of the requests served after the one that is pending at time t_2. Finally, for the i-th application to receive at least the minimum amount of service that the real system is claimed to guarantee during $[t_1, t_2]$, it must *ask for* such service, i.e., the sum of the sizes of the requests issued during $[t_1, t_2]$, plus $Q_i(a_i^{j-})$, must be no lower than this amount of service. Thanks to these considerations, to simplify the proofs, and without losing generality, we assume that, starting from time a_i^j, the i-th application is continuously backlogged and issues all its next requests asynchronously and back-to-back.

Using Lemma A.3, we can write

$$\begin{aligned}
W_i(t_1, t_2) &= \\
W_i^I(t_2) - \mathrm{lag}_i(t_2) - (W_i^I(t_1) - \mathrm{lag}_i(t_1)) &\geq \\
W_i^I(t_2) - B_{max} - L_{max} - W_i^I(t_1) + & \\
+ \min(\phi_i(V_i(t_1) - F_i(t_1)), 0) &\geq \\
\phi_i \left(V(t_2) - V_i(t_1)\right) + \min(\phi_i(V_i(t_1) - F_i(t_1)), 0) + & \\
- B_{max} - L_{max}, &
\end{aligned} \tag{22}$$

where the last inequality follows from the fact that, since the i-th application is necessarily continuously backlogged also in the ideal system, then $W_i^I(t_1, t_2) \geq \phi_i \left(V(t_2) - V_i(t_1)\right)$. For any set of values of the other quantities, the rightmost term is lower if $F_i(t_1) > V_i(t_1)$. Hence, thanks to (14)

$$\begin{aligned}
W_i(t_1, t_2) &\geq \\
\phi_i \left(V(t_2) - V_i(t_1) + V_i(t_1) - F_i(t_1)\right) - B_{max} - L_{max} &= \\
\phi_i \left(V(t_2) - F_i(t_1)\right) - B_{max} - L_{max} &\geq \\
\phi_i \left(V(t_2) - V(t_1) - \frac{B_{i,max}}{\phi_i}\right) - B_{max} - L_{max} &\geq \\
\phi_i W(t_1, t_2) - B_{i,max} - B_{max} - L_{max}. &
\end{aligned} \tag{23}$$

Before concluding the proof, it is important to note that, as anticipated in Section 2.2.2, the $B_{i,max}$ component in the rightmost term is a consequence of the real system being in advance with respect to the ideal system in serving the i-th application at time a_i^j. On the contrary, this component is equal to 0 if R_i^j may arrive only after the minimum completion time guaranteed by the ideal system to the previous request R_i^{j-1}. □

A.2. Time Guarantees

THEOREM 2.2. *Given a request R_i^j, let $t_1 \leq a_i^j$ be a generic time instant such that the i-th 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 i-th application during $[t_1, a_i^j + T_{FIFO})$ plus L_i^j. The following inequality holds:*

$$\begin{aligned} c_i^j - t_1 \quad \leq \\ \frac{Q_i(t_1^-) + A_i(t_1, a_i^j + T_{FIFO}) + (B_i^l - L_i^j) + B_{i,max}}{\phi_i T_{agg}} + \\ + \frac{B_{max} + B_{i,max} + L_{max}}{T_{agg}}, \end{aligned} \tag{24}$$

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

PROOF. Let c_i^j and $c_i^{I,m}$ be the completion time of R_i^j in the real system and of R_i^m in the ideal system, respectively. We have that:

$$\begin{aligned} W(t_1, c_i^{I,m}) \leq V(t_1, c_i^{I,m}) = V_i(t_1) - V_i(t_1) + V(t_1, c_i^{I,m}) \quad = \\ V_i(t_1) - V_i(t_1) + V(c_i^{I,m}) - V(t_1) \quad = \\ \left(V_i(t_1) - V(t_1)\right) + \left(V(c_i^{I,m}) - V_i(t_1)\right) \quad \leq \\ \left(V_i(t_1) - V(t_1)\right) + \left(V_i(c_i^{I,m}) - V_i(t_1)\right) \quad \leq \\ \left(V_i(t_1) - V(t_1)\right) + \frac{W_i(t_1, c_i^{I,m})}{\phi_i}, \end{aligned} \tag{25}$$

where the last inequality follows from the fact that the i-th application is continuously backlogged (in the ideal system) during $[t_1, c_i^{I,m}]$.

Let $Q^I(a_i^{j-})$ be the backlog of the i-th application in the ideal system at time a_i^{j-}. We have that $W_i^I(t_1, c_i^{I,m}) \leq Q^I(t_1^-) + A_i(t_1, a_i^j + T_{FIFO})$. Hence, since the ideal system has a constant throughput equal to T_{agg}, we have that

$$c_i^{I,m} - t_1 \leq \frac{\phi_i \left(V_i(t_1) - V(t_1)\right) + Q^I(t_1^-) + A_i(t_1, a_i^j + T_{FIFO})}{\phi_i T_{agg}}. \tag{26}$$

As a first step to derive the thesis from the above inequality, we can consider that, thanks to Lemma A.3, $Q^I(t_1^-) = Q(t_1^-) - \mathrm{lag}_i(t_1) \leq Q(t_1^-) - \min\left(\phi_i(V_i(t_1) - F_i(t_1)), 0\right)$. Substituting this inequality in (26), we get

$$\begin{aligned} c_i^{I,m} - t_1 \leq \frac{\phi_i \left(V_i(t_1) - V(t_1)\right) + Q(t_1^-)}{\phi_i T_{agg}} + \\ + \frac{-\min\left(\phi_i(V_i(t_1) - F_i(t_1))\right), 0) + A_i(t_1, a_i^j + T_{FIFO})}{\phi_i T_{agg}}. \end{aligned} \tag{27}$$

For any set of values of the other quantities, the rightmost term is higher if $F_i(t_1) \geq V_i(t_1)$. Hence, thanks to (14),

$$\begin{aligned} c_i^{I,m} - t_1 &\leq \\ \frac{\phi_i \left(V_i(t_1) - V(t_1)\right) + Q(t_1^-) - \phi_i(V_i(t_1) - F_i(t_1)) + A_i(t_1, a_i^j + T_{FIFO})}{\phi_i T_{agg}} &= \\ \frac{\phi_i \left(F_i(t_1) - V(t_1)\right) + Q(t_1^-) + A_i(t_1, a_i^j + T_{FIFO})}{\phi_i T_{agg}} &\leq \\ \frac{B_{i,max} + Q(t_1^-) + A_i(t_1, a_i^j + T_{FIFO})}{\phi_i T_{agg}}. & \end{aligned} \tag{28}$$

The thesis follows from considering that, thanks to Lemma A.2, $c_i^j - c_i^{I,m} \leq \frac{B_i^l - L_i^j}{\phi_i T_{agg}} + \frac{B_{max} + L_{max}}{T_{agg}}$. Regarding the $B_{i,max}$ component, the same considerations about the arrival pattern reported at the end of the proof of Theorem 2.1 apply. □

APPENDIX B

Proofs of QFQ Properties

We prove here both the properties of the data structure used in Section 3.1, and the B/T-WFI of QFQ. In this chapter, we explicitly indicate the time at which any timestamp is computed to avoid ambiguity, and we assume that the various quantities ($V(t)$, $S^k(t)$, ...) are computed as described in the QFQ algorithm. For notational convenience, we avoid writing $f(t_c^+)$ if $f(t)$ is continuous at time t_c. To further simplify the notation, if the function is discontinuous at a time instant t_d, we assume, without losing generality, that $f(t_d) \equiv \lim_{t \to t_d^-} f(t)$, i.e., that the function is left-continuous. Finally, recall that all the quantities used hereafter are also reported in Table 1.

We start with Section B.1, which contains some relations we will need in the rest of the chapter, then in Section B.2 we proof the correctness of the scheduler showing the properties of the data structures it uses; finally we prove its service properties in Section B.3.

B.1. Preliminary Results

We define the following two notations for convenience:

$$\lfloor x \rfloor_{\sigma_i} \equiv \left\lfloor \frac{x}{\sigma_i} \right\rfloor \sigma_i \, , \lceil x \rceil_{\sigma_i} \equiv \left\lceil \frac{x}{\sigma_i} \right\rceil \sigma_i .$$

For any positive quantity $y < x + \sigma_i$, we have

$$\lfloor y \rfloor_{\sigma_i} \leq \lceil x \rceil_{\sigma_i} . \tag{29}$$

In fact, x can be written as $x = n\sigma_i + \delta$, with $0 \leq \delta < \sigma_i$. If $\delta = 0$ then $y < (n+1)\sigma_1 \implies \lfloor y \rfloor_{\sigma_i} \leq n\sigma_i$, $\lceil x \rceil_{\sigma_i} = n\sigma_i$, and the thesis holds; if $\delta > 0$ then $\lfloor y \rfloor_{\sigma_i} \leq (n+1)\sigma_i$, $\lceil x \rceil_{\sigma_i} = (n+1)\sigma_i$, and the thesis holds too.

B.1.1. Group GBT Under QFQ. We start by proving per-group upper bounds for $S_i(t) - V(t)$ (in Theorem B.1) and for $V(t) - F_i(t)$ (Theorem B.2, supported by the two long Lemmas B.1 and B.2). The two bounds represent a group-based variant of the *Globally Bounded Timestamp* (GBT) property, normally defined for the flow timestamps in an exact virtual time-based scheduler. Lemmas B.1 and B.2 are an adapted version of the ones in [**Kar**], repeated here for convenience, with permission from the author.

We will use these bounds to prove both the properties of the data structure and the B/T-WFI of QFQ.

THEOREM B.1 (Upper bound for $S_i(t) - V(t)$). *For any backlogged group i and $\forall t$ the following condition holds:*

$$S_i(t) \leq \left\lceil \frac{V(t)}{\sigma_i} \right\rceil \sigma_i = \lceil V(t) \rceil_{\sigma_i}. \tag{30}$$

PROOF. By definition (8), at any time t and for any group i, $S_i(t)$ is an integer multiple of σ_i and, for any backlogged flow k of the group, $S_i(t) \leq \hat{S}^k(t) = \lfloor S^k(t) \rfloor_{\sigma_i}$. It follows that, if $S^k(t) < V(t) + 2\sigma_i$, then (30) trivially holds. Hence, to prove (30) we actually prove the latter, i.e., that $S^k(t) < V(t) + 2\sigma_i$, and to prove it we consider only a generic time instant t_1 at which a generic packet for flow k is enqueued/dequeued, as this is the only event upon which which $S^k(t)$ may increase.

According to (2), either $S^k(t_1^+) = V(t_1)$, in which case the packet is enqueued and the thesis trivially holds, or $S^k(t_1^+) = F^k(t_1)$. In this case flow k must have had a packet previously dequeued at time $t_p < t_1$.

When the packet was dequeued at t_p flow k was certainly eligible, and $V(t)$ is immediately incremented after the dequeue at t_p, so we have $F^k(t_1) = S^k(t_p^+) = S^k(t_p) + l^k(t_p)/\phi^k \leq V(t_p) + \sigma_i + l^k(t_p)/\phi^k \leq V(t_p) + 2\sigma_i < V(t_p^+) + 2\sigma_i$, which proves the thesis. □

LEMMA B.1. *Let $I(t) = \{k : k \in B(t), S^k(t) \geq V(t)\}$ be a subset of flows. Given a constant V', $\forall t : V(t) \leq V'$ we have:*

$$\sum_{k \in I(t)} \left(l^k(t) + \phi^k [V' - F^k(t)] \right) \leq V' - V(t), \tag{31}$$

where $l^k(t)$ is the size of the first packet in the queue for flow k at time t.

PROOF. By definition, $l^k(t) = \phi^k [F^k(t) - S^k(t)]$. Thus, for flows in set $I(t)$ we have $l^k(t) \leq \phi^k [F^k(t) - V(t)]$. Therefore, with simple algebraic passages:

$$\sum_{k \in I(t)} \left\{ l^k(t) + \phi^k [V(t) - F^k(t)] \right\} \leq 0 \tag{32}$$

$$\sum_{k \in I(t)} \left\{ l^k(t) + \phi^k [V(t) - V'] + \phi^k [V' - F^k(t)] \right\} \leq 0 \tag{33}$$

$$\sum_{k \in I(t)} \left\{ l^k(t) + \phi^k [V' - F^k(t)] \right\} \leq \sum_{k \in I(t)} \phi^k [V' - V(t)] \tag{34}$$

$$\sum_{k \in I(t)} \left\{ l^k(t) + \phi^k [V' - F^k(t)] \right\} \leq V' - V(t), \tag{35}$$

where the last passage uses $\sum_{k \in I} \phi^k \leq 1$. □

LEMMA B.2. *Let $X(t, M) \equiv \{k : \hat{F}^k(t) \leq M\}$ be a set of flows. Given a constant V', we have that $\forall t : L + V' \geq V(t)$:*

$$\sum_{k \in X(t, V')} \left(l^k(t) + \phi^k [V' - F^k(t)] \right) \leq L + V' - V(t). \tag{36}$$

PROOF. The proof is by induction over those events that change the terms in (36): Packet enqueues for idle flows, packet dequeues and virtual time jumps. The base case where X is empty is true by assumption. For the inductive proof, we assume (36) to hold at some time t_1.

Packet enqueue for an idle flow: Say a packet of size l_1 of the idle flow k arrives at time t_1. $V(t)$ does not change on packet arrivals except for virtual time jumps, that are dealt with later.
If after the enqueue of the new packet $k \notin X(t_1^+, V')$, i.e., $\hat{F}^k(t_1^+) > V'$, we must consider two sub-cases. First, if $k \notin X(t_1, V')$ nothing changes. Second, if $k \in X(t_1, V')$ the positive component $\phi^k[V' - F^k(t_1)]$ is removed from the sum, so the left hand side of (36) decreases. In both sub-cases the lemma holds. The remaining case is if $k \in X(t_1^+, V')$. Since $\hat{F}^k(t_1^+) > \hat{F}^k(t_1)$, this implies $k \in X(t_1, V')$. In this case $l^k(t)$ is incremented by l_1, but $F^k(t)$ is incremented by l_1/ϕ^k, so the left hand side of (36) remains unchanged and the lemma holds.

Virtual time jump: After a virtual time jump, all backlogged flows have $S^k(t_1^+) \geq \hat{S}^k(t_1^+) \geq V(t_1^+)$. With regard to idle flows, we assume that their virtual start and finish times are pushed to $V(t_1^+)$. By doing so we do not lose generality, as the virtual start times of these flows will be lower-bounded by $V(t)$ when they become backlogged (again). Besides, it is easy to see that pushing up their virtual finish times may only let the left side of (36) decrease. In the end $S^k(t_1^+) \geq V(t_1^+)$ for all flows and, if $V' \geq V(t_1^+)$ then Lemma B.1 applies and the lemma holds. For other V' in $[V(t_1^+) - L, V(t_1^+)[$, the additional L term in (36) absorbs any decrement on the right hand side. Therefore, the lemma holds.

Packet dequeue: Flow k receives service at time t_1 for its head packet of size $l^k(t_1)$. We have to distinguish two cases, depending on V' and $\hat{F}^k(t_1)$.

(1) $V' \geq \hat{F}^k(t_1)$. $V(t)$ is incremented exactly by $l^k(t_1)$, so the right side of (36) decreases exactly by $l^k(t_1)$. With regard to the left side, the variation of $l^k(t)$ can be seen as the result of first decreasing by $l^k(t_1)$, which balances the above decrement of $V(t)$, and then increasing by $l^k(t_1^+)$, which is in turn balanced by incrementing $F^k(t)$ by $\frac{l^k(t_1^+)}{\phi^k}$. Hence the lemma holds.

(2) $V' < \hat{F}^k(t_1)$. In this case all flows $h \in X(t_1, V')$ have $\hat{F}^h(t_1) < \hat{F}^k(t_1)$, so they must have been ineligible according to their rounded start time, otherwise the current flow k would have not been chosen. Therefore, $V(t_1) < \hat{S}^h(t_1) \leq S^h(t_1)$ for all flows in $X(t_1, V')$. Lemma B.1 applies then for all $V' \geq V(t_1)$, i.e.,

$$\sum_{k \in X(t_1^+, V')} \left(l^k(t_1) + \phi^k[V' - F^k(t_1)] \right) \leq V' - V(t_1). \tag{37}$$

Because $V(t_1^+) = V(t_1) + l^k$ and we assume $L + V' \geq V(t_1^+)$ after service, we only need to consider V' with $L + V' \geq V(t_1^+) + l^k$ before service. Therefore

$$V' - V(t_1) \leq (L - l^k) + V' - (V(t_1^+) - l^k) = L + V' - V(t_1^+) \tag{38}$$

and the lemma holds after service.

This concludes the proof. □

THEOREM B.2 (Upper bound for $V(t) - F_i(t)$). *For any backlogged group i*

$$V(t) \leq F_i(t) + L. \tag{39}$$

PROOF. To prove the thesis we will actually prove the more general inequality $V(t) \leq \hat{F}^k(t) + L$ for a generic flow k of group i. The proof is by contradiction. The only event that could lead to a violation of the assumption is serving a packet. Assume that at $t = t1 : V(t_1) = V_1$ the lemma holds. A packet p with rounded finish time $\hat{F}_1$ and length l_p is served and afterwards at time $t_2 : V(t_2) = V_2$,

there is a packet q with finish time F_2, such that $\hat{F}_2 + L < V_2$. Denote with $\hat{S}_1$ and $\hat{S}_2$ the corresponding start times. We need to distinguish three cases.

(1) Packet q is eligible at time t_1 according to its rounded start time. Then, $\hat{F}_2 \geq \hat{F}_1$ (both packets were eligible at V_1 and p was chosen). Applying Lemma B.2 with $t = t_1$ and $V' = \hat{F}_2$ results in

$$\sum_{k \in X(t_1, \hat{F}_2)} l^k(t_1) + \sum_{k \in X(t_1, \hat{F}_2)} (\hat{F}_2 - F^k(t_1))\phi^k \leq L + \hat{F}_2 - V(t_1). \tag{40}$$

Because $F^k(t) \leq \hat{F}^k(t)$, the second term on the left side of the inequality is non-negative and therefore

$$l_p \leq \sum_{k : \hat{F}^k(t) \leq \hat{F}_2} l^k(t) \leq L + \hat{F}_2 - V_1 \tag{41}$$

$$V_2 - V_1 \leq L + \hat{F}_2 - V_1 \tag{42}$$

$$V_2 \leq \hat{F}_2 + L \tag{43}$$

The step from (41) to (42) uses $V_1 + l_p = V_2$.

(2) Packet q is not eligible at V_1 according to its rounded start time, but becomes eligible between V_1 and V_2. Then, $\hat{S}_2 \geq V_1$. Virtual time advances by at most L and therefore:

$$\hat{F}_2 \geq \hat{S}_2 \geq V_1 \geq V_2 - L. \tag{44}$$

(3) Packet q is not eligible according to its rounded start time after service to p, therefore V_2 is reached by a virtual time jump before q can be served. In this case:

$$\hat{F}_2 \geq \hat{S}_2 \geq V_2 \geq V_2 - L. \tag{45}$$

This concludes the proof. □

B.2. Data Structures' Properties

We can now prove the theorems used in Section 3.1 considering the only two events that can change the state of the scheduler, namely packet enqueue and packet dequeue. We start from Theorem B.3, which gives the per-group slot occupancy.

THEOREM B.3. *At all times, only the first* $2 + \left\lceil \frac{L}{\sigma_i} \right\rceil$ *consecutive slots beginning from the head slot of a group may be non empty.*

PROOF. Consider a generic flow k belonging to a group i. A new virtual start time may be assigned to the flow (only) as a consequence of the enqueueing/dequeueing of a new packet $p^{k,l}$ at a time instant t_p. As in the proof of Lemma B.1, from (2) $S^k(t_p^+)$ may be equal to either (a) $V(t_p)$, or (b) $F^k(t_p)$, where we assume $F^k(t_p) = 0$ if $p^{k,l}$ is the first packet of the flow to be enqueued/dequeued.

In the first case, according to (39), $S^k(t_p^+) = V(t_p) \leq F_i(t_p) + L \leq S_i(t_p) + 2\sigma_i + L \leq S_i(t_p) + 2\sigma_i + \left\lceil \frac{L}{\sigma_i} \right\rceil \sigma_i$. In the second case, neglecting the trivial sub-case $F^k(s^{k,l-1+}) = 0$, we can consider that flow k had to be a head flow when $p^{k,l-1}$ was served. Hence, according to (8), $S^k(t_p) < S_i(t_p) + \sigma_i$. From (2), this implies $S^k(t_p^+) = F^k(t_p) < S_i(t_p) + 2\sigma_i \leq S_i(t_p) + 2\sigma_i$.

Considering both cases, it follows that, $\forall t\ S^k(t) - S_i(t) < (2 + \left\lceil \frac{L}{\sigma_i} \right\rceil)\sigma_i$, i.e., that at any time the virtual start times of all the backlogged flows of a group may belong only to the $2 + \left\lceil \frac{L}{\sigma_i} \right\rceil$ consecutive slots beginning from the one the head slot queue is associated to, which proves the thesis. □

Using the following lemma, we want now to prove that **ER** is ordered by virtual finish times.

LEMMA B.3. *Let $\overline{t}$ be the time instant at which a previously idle group i becomes backlogged, or at which the group, previously ineligible, becomes eligible, or finally at which the virtual finish time of the group decreases. We have that $F_h(\overline{t}) \leq F_i(\overline{t}^+)$ for any backlogged group h with $h < i$.*

PROOF. For $F_i(t)$ to decrease, $S_i(t)$ must decrease as well. According to the `enqueue()` and `dequeue()`, this can happen only in consequence of the enqueueing of a packet of an empty flow of the group. As this is exactly the same event that may cause a group to become backlogged, then, from (2) we have $S_i(\overline{t}^+) \geq \lfloor V(\overline{t}) \rfloor_{\sigma_i}$ both if the group become backlogged and if $F_i(t)$ decreases. Substituting this inequality, which finally holds also if the group becomes eligible at time $\overline{t}$, and (30) in the following difference we get:

$$\begin{aligned} F_h(\overline{t}) - F_i(\overline{t}^+) &= \\ S_h(\overline{t}) + 2\sigma_h - S_i(\overline{t}^+) - 2\sigma_i &= \\ S_h(\overline{t}) - S_i(\overline{t}^+) + 2\sigma_h - 2\sigma_i &\leq \\ \lceil V(\overline{t}) \rceil_{\sigma_h} - \lfloor V(\overline{t}) \rfloor_{\sigma_i} + 2\sigma_h - 2\sigma_i &\leq \\ \lceil V(\overline{t}) \rceil_{\sigma_i} - \lfloor V(\overline{t}) \rfloor_{\sigma_i} + 2\sigma_h - 2\sigma_i &\leq \\ \sigma_i + 2\sigma_h - 2\sigma_i &= \\ 2\sigma_h - \sigma_i &\leq 0, \end{aligned} \tag{46}$$

where $\lceil V(\overline{t}) \rceil_{\sigma_h} \leq \lceil V(\overline{t}) \rceil_{\sigma_i}$ and the last inequality follow from that, as $i > h$, $\sigma_i \geq 2\sigma_h$. □

The following theorem guarantees that **ER** is always ordered by virtual finish times. Then it guarantees that this order is never broken when one or more groups are inserted into it during QFQ operation.

THEOREM B.4. *Set* **ER** *is ordered by group virtual finish time.*

PROOF. We will prove the thesis by induction. In the base case $\mathbf{ER} = \emptyset$ the thesis trivially holds. The ordering of **ER** may change only when one or more groups enter the set. This can happen as a consequence of:

(1) A group entering **ER** as it becomes backlogged.
(2) One or more groups moving from **IR** to **ER**.
(3) One or more groups moving from **EB** to **ER**.

Let i be a group entering **ER** at time t_1 for one of the above three reasons, and let the thesis hold before time t_1.

In the first case, thanks to Lemma B.3, $F_i(t_1^+)$ is not lower than the virtual finish times of the groups in **ER** with lower index. By definition of **ER**, $F_i(t_1^+)$ is also not higher than the virtual finish times of the groups in **ER** with higher index.

In the second case, given a group $h \in \mathbf{ER}$ with $h < i$, $S_i(t_1) \geq S_h(t_1)$ because either group h was already in **ER** before time t_1, or group h belonged to **IR**, which is ordered by virtual start times according to Section 3.1.3, item 2. This implies $F_i(t_1) \geq F_h(t_1)$ because $\sigma_i \geq 2\sigma_h$. By definition of **IR**, $F_i(t_1)$ is also not higher than the virtual finish times of the groups in **ER** with higher index.

In the third case, since group i is not blocked any more, $F_i(t_1)$ is not higher than the virtual finish times of the groups in **ER** with higher index. With regard to the groups with lower index than i, for group i to be blocked before time t_1 there had to be a group $b \in \mathbf{ER}$ with $b > i$ and $F_b(t_1) < F_i(t_1)$. Since we assume that **ER** is ordered by virtual finish time before time t_1, then $F_b(t_1)$, and hence $F_i(t_1)$ is not lower than the virtual finish times of all the lower index groups in **ER**. □

To prove that **EB** enjoys the same order property as **ER**, we need first a further lemma. The validity of the lemma depends on the timestamp *back-shifting* performed under QFQ when inserting a newly backlogged group into **EB**. Hence this is the right moment to explain in detail this operation. When an idle group i becomes blocked after enqueuing a packet of a flow k at time t_p, the timestamps of flow k are not updated using the following variant of (2):

$$\begin{aligned} S^k(t_p^+) &\leftarrow \max\left[\min(V(t_p), F_b(t_p)), F^k(t_p)\right] \\ F^k(t_p^+) &\leftarrow S^k + l^k(t_p^+)/\phi^k, \end{aligned} \tag{47}$$

where b is the lowest order group in **ER** such that $b > i$. Basically, with respect to the exact formula, $F_b(t_p)$ is used instead of $V(t_p)$ if $V(t_p) > F_b(t_p)$. This is done because otherwise the ordering by virtual finish time in **EB** may be broken. It would be easy to show that this would happen if an idle group becomes blocked when $V(t)$ is too higher than the virtual finish time of some other blocked group $h < i$.

With regard to worst-case service guarantees, in case $V(t_p) > F_b(t_p)$ in (47), group i just benefits from the back-shifting, whereas the guarantees of the other flows are unaffected. To prove it, consider that the guarantees provided to any flow do not depend on the actual arrival time of the packets of the other flows. Hence one can still "move" a pair of timestamps backwards, provided that this does not lead to an inconsistent schedule, i.e., provided that the resulting schedule is the same as if the packet would have actually arrived at the time that would have lead to that value of the pair of timestamps according to the exact formulas. And this is what happens using (47), for the following reason. Should the packet that lets group i become backlogged had arrived at a time instant $\overline{t}_p \leq t_p$ at which $V(\overline{t}_p) = F_b(t_p)$, group i would have however got a virtual start time higher than $F_b(t_p)$. In addition, since $V(\overline{t_p}) = F_b(t_p)$, then group b must have been backlogged at time $\overline{t_p} \leq t_p$ to have a virtual finish time equal to $F_b(t_p)$ at time t_p. In the end group i would not have been served before group b, exacly as it happens in the schedule resulting from timestamping group i with (47) at time t_p.

We can now prove the intermediate lemma we need to finally prove the ordering in **EB**.

LEMMA B.4. *If a pair of groups h and i with $h < i$ are blocked at a generic time instant t_2, then $S_h(t_2) \leq F_i(t_2)$.*

PROOF. We consider two alternative cases. The first is that $S_h(t_2)$ has been last updated at a time instant $t_1 \leq t_2$ using (47). The second is that, according to (2) and (8) there are at least one head flow k of group h and a time instant $t_1 \leq t_2$ such that $S_h(t_2) = \lfloor F^k(t_1) \rfloor_{\sigma_h}$.

In the first case we have $S_b(t_2) \leq F_b(t_1)$, where b is the lowest order group in **ER** such that $b > h$. We can consider two sub-cases. First, group i is already backlogged and eligible at time t_1. It follows that, if $i \geq b$ then $F_i(t_1) \geq F_b(t_1)$. Otherwise, from the definition of b, group i is necessarily blocked, and $F_i(t_1) > F_b(t_1)$ must hold again for group b not to be blocked. In the end, regardless of whether group i is ready or blocked, $F_i(t_2) \geq F_i(t_1) > F_b(t_1) = S_b(t_2)$ and the thesis holds. In the other sub-case, i.e., group i is not ready and eligible at time t_1, thanks to Lemma B.3 group i cannot happen to have a virtual finish time lower than $F_b(t_1)$ during $(t_1, t_2]$. Hence $F_i(t_2) \geq F_h(t_1) = F_h(t_2) > S_h(t_2)$ and the thesis holds.

In the other case, i.e., $S_h(t_2) = \lfloor F^k(t_1) \rfloor_{\sigma_h}$, we prove the thesis by contradiction. Suppose that $S_h(t_2) > F_i(t_2)$. Flow k must have necessarily been served with $F^k(t_0) = F^k(t_1)$ at some time $t_0 \leq t_1$. In addition, for $S_h(t_2) > F_i(t_2)$ to hold, $F^k(t_1) > F_i(t_2)$ and hence $F^k(t_0) > F_i(t_2)$ should hold as well. As flow k had to be a head flow at time t_0, it would follow that

$$F_h(t_0) \geq F^k(t_0) > F_i(t_2). \tag{48}$$

We consider two cases. First, group i is backlogged at time t_0. If $F_i(t_0) < F_h(t_0)$, then $S_i(t_0) = F_i(t_0) - 2\sigma_i < F_h(t_0) - 2\sigma_i < S_h(t_0)$, because $\sigma_i > \sigma_h$. Hence, both group h and i would be eligible, and group h could not be served at time t_0. In follows that $F_i(t_0) \geq F_h(t_0)$ should hold. This inequality and (48) would imply $F_i(t_0) > F_i(t_2)$. Should not $F_i(t)$ decrease during $[t_0, t_2]$, the absurd $F_i(t_2) > F_i(t_2)$ would follow. But, from `enqueue()` and `dequeue()` it follows that the only event that can let $F_i(t)$ decrease is the enqueueing of a packet of an idle flow of group i that causes $S_i(t)$ to decrease (lines 12-18 of `enqueue`). Let $F_{i,min}$ be the minimum value that $F_i(t)$ may assume in consequence of this event. Since $\forall t \in [t_0, t_2]\ V(t) \geq S_h(t_0)$, according to (2), (8) and (48), $F_{i,min} \geq \lfloor S_h(t_0) \rfloor_{\sigma_i} + 2\sigma_i \geq S_h(t_0) - \sigma_h + 2\sigma_i = F_h(t_0) - 3\sigma_h + 2\sigma_i > F_h(t_0) > F_i(t_2)$, which again would imply the absurd $F_i(t_2) > F_i(t_2)$.

The second case is that group i is not backlogged at time t_0. As the event that would let the group become backlogged after time t_0 is the same that might have let $F_i(t)$ decrease in the other case, then, using the same arguments as above, we would get the same absurd.

In the end, $S_h(t_2) \leq F_i(t_2)$ must hold. □

The following theorem guarantees that **EB** is always ordered by virtual finish time (hence, as previously proven for **ER** this order is never broken during QFQ operations).

THEOREM B.5. *Set* **EB** *is ordered by group virtual finish time.*

PROOF. We will prove the thesis by induction. In the base case $\mathbf{EB} = \emptyset$ the thesis trivially holds. The only event upon which the the ordering of **EB** may change is when one or more groups enter the set. The three events that may cause a group to become blocked are:

(1) The enqueueing/dequeueing of a packet of a flow of an idle group $j > i$, which lets group j get a lower virtual finish time than group i (groups with lower order than i can never block group i).
(2) The enqueueing/dequeueing of a packet of a flow of group i itself, which lets the virtual finish time of group i become higher than the virtual finish of some higher order group.
(3) The growth of $V(t)$, which causes one or more groups to move from **IB** to **EB**.

With regard to the first event, it is worth noting that group j can cause group i to become blocked only if group j becomes backlogged or if $F_j(t)$ decreases. Let t_1 be the time instant at which one of these two events occurs and such that **EB** is ordered up to time t_1. Thanks to Lemma B.3, $F_i(t_1) \leq F_j(t_1^+)$ and hence the event cannot let group i become blocked.

Suppose now that, at time t_1, group i enters **EB** as a consequence of either a packet of a flow of the group being enqueued/dequeued or the growth of $V(t)$. We will prove that, given two any blocked groups $h < i$ and $j > i$, $F_h(t_1) \leq F_i(t_1^+)$ and $F_i(t_1^+) \leq F_j(t_1)$ hold (where $F_i(t_1^+) = F_i(t_1)$ in case group i enters **EB** from **IB**).

With regard to a blocked group $h < i$, if group i enters **EB** as a consequence of a packet enqueue/dequeue, then, from Lemma B.4 and the fact that, as $F_i(t)$ is an integer multiple of σ_i, $F_i(t_1^+) \geq F_i(t_1) + \sigma_i$, we have

$$\begin{aligned} F_i(t_1^+) - F_h(t_1) &\geq \\ F_i(t_1) + \sigma_i - S_h(t_1) - 2\sigma_h &\geq \\ F_i(t_1) + \sigma_i - F_i(t_1) - 2\sigma_h &\geq \\ \sigma_i - 2\sigma_h &\geq 0, \end{aligned} \tag{49}$$

where the last inequality follows from $\sigma_i \geq 2\sigma_h$. On the other hand, if group i enters **EB** from **IB**, then $S_i(t_1) \geq S_h(t_1)$ because either group h was already eligible before time t_1, or group h belonged to **IB**, which is ordered by virtual start time according to Section 3.1.3, item 2. This implies $F_i(t_1) \geq F_h(t_1)$ because $\sigma_i \geq 2\sigma_h$.

With regard to a blocked group $j > i$, let $b > j > i$ be the highest order group that is blocking group j at time $\overline{t}$. Independently of the reason why group i enters **EB**, from Lemma B.4 we have

$$S_i(\overline{t}^+) \leq F_b(\overline{t}) \leq F_j(\overline{t}) - \sigma_j, \tag{50}$$

where the last inequality follows from $F_b(\overline{t}) < F_j(\overline{t})$ and the fact that both $F_j(\overline{t})$ and $F_b(\overline{t})$ are integer multiples of σ_j. Substituting (50) in what follows:

$$\begin{aligned} F_i(\overline{t}^+) &= \\ S_i(\overline{t}^+) + 2\sigma_i &\leq \\ F_j(\overline{t}) - \sigma_j + 2\sigma_i &\leq \\ F_j(\overline{t}) - 2\sigma_i + 2\sigma_i, & \end{aligned} \tag{51}$$

where the penultimate inequality follows from that, since $j > i$, $\sigma_j \geq 2\sigma_i$. □

Finally, we can prove the theorem that allows QFQ to quickly choose the groups to move from **EB/IB** to **ER/IR**.

THEOREM B.6 (Group unblocking). *Let i be the group that would be served upon the next packet dequeue at time $\overline{t}$, and assume that there is no group $j : j > i, F_j(\overline{t}) = F_i(\overline{t})$; in this case, if group i is actually served and $F_i(\overline{t}^+) > F_i(\overline{t})$ or if group i becomes idle at time $\overline{t}$, then all and only the groups in* **EB/IB** *and with order lower than i must be moved into* **ER/IR**.

PROOF. To prove the thesis, we first prove that group i is the only group that can block a group $h < i$. The proof is by contradiction. Suppose for a moment that a group $j > i$ blocks group h. Since

$F_i(\overline{t}) < F_j(\overline{t})$ must hold for group i not to be blocked, and both $F_i(\overline{t})$ and $F_j(\overline{t})$ are integer multiples of σ_i, then

$$F_i(\overline{t}) \leq F_j(\overline{t}) - \sigma_i. \tag{52}$$

Combining this inequality with Lemma B.4, we get $S_h(\overline{t}) \leq F_j(\overline{t}) - \sigma_i$ and hence, considering that $\sigma_i \geq 2\sigma_h$, $F_h(\overline{t}) = S_h(\overline{t}) + 2\sigma_h \leq F_j(\overline{t}) - \sigma_i + 2\sigma_h \leq F_j(\overline{t})$. This contradicts the fact that group j blocks group h.

As a consequence, if $F_i(t)$ increases, then, thanks to (49) and (51), all and only the blocked groups $h < i$ become ready. The same happens if group i becomes idle as a consequence of a packet dequeue. □

B.3. Service Properties

We have already stated Theorem 3.1 and Theorem 3.2, that give the B-WFI and T-WFI for QFQ, in this section we prove them.

THEOREM 3.1. *For a flow k belonging to group i QFQ guarantees*

$$\text{B-WFI}^k = 3\phi^k \sigma_i + 2\phi^k L. \tag{53}$$

PROOF. We consider two cases. First, flow k is eligible at time t_1. In this case, we consider that, given the virtual time $V^k(t)$ of flow k in the real system, $V^k(t_1) \leq F^k(t_1)$ and $V^k(t_2) \geq S^k(t_2)$. Hence, considering also that, thanks to (39), $S_i(t_2) = F_i(t_2) - 2\sigma_i > V(t_2) - L - 2\sigma_i$, we have:

$$\begin{aligned}
W^k(t_1,t_2) &= \\
\phi^k V^k(t_1,t_2) &= \\
\phi^k(S^k(t_2) - V^k(t_1)) &\geq \\
\phi^k(S^k(t_2) - F^k(t_1)) &\geq \\
\phi^k(S_i(t_2) - F^k(t_1)) &> \\
\phi^k(S_i(t_2) - (S^k(t_1) + \sigma_i)) &> \\
\phi^k(V(t_2) - L - 2\sigma_i - (V(t_1) + \sigma_i)) &= \\
\phi^k(V(t_2) - V(t_1) - L - 3\sigma_i) &= \\
\phi^k(V(t_2) - V(t_1)) - \phi^k L - 3\phi^k \sigma_i &> \\
\phi^k W(t_1,t_2) - 2\phi^k L - 3\phi^k \sigma_i,
\end{aligned} \tag{54}$$

where the last inequality follows from the fact that, because of the immediate increment of $V(t)$ as a packet is dequeued (see `updateV()`), $V(t_2) - V(t_1) \geq W(t_1,t_2) - L$.

Second, flow k is not eligible at time t_1. This implies that the flow virtual time is exactly equal to $S^k(t_1)$ at time t_1 and hence, considering that, $S^k(t_1) \leq V(t_1) + \sigma_i$, we have:

$$\begin{aligned} W^k(t_1,t_2) &\geq \\ \phi^k(S^k(t_2) - S^k(t_1)) &\geq \\ \phi^k(S_i(t_2) - S^k(t_1)) &> \\ \phi^k(V(t_2) - L - 2\sigma_i - S^k(t_1)) &> \\ \phi^k(V(t_2) - L - 2\sigma_i - (V(t_1) + \sigma_i) &> \\ \phi^k(V(t_2) - V(t_1) - L - 3\sigma_i) &> \\ \phi^k W(t_1,t_2) - 2\phi^k L - 3\phi^k \sigma_i. & \end{aligned} \tag{55}$$

□

THEOREM 3.2. *For a flow k belonging to group i, and a link with constant rate R, QFQ guarantees*

$$\text{T-WFI}^k = \left(3 \left\lceil \frac{L^k}{\phi^k} \right\rceil + 2L \right) \frac{1}{R}. \tag{56}$$

PROOF. Assume a generic packet arriving at t_a and completing service at t_c. Let $Q^k(t_a^+)$ be the backlog of flow k just after the arrival of the packet. Because of the immediate increment of $V(t)$ upon packet dequeue we have $V(t_a^+, t_c) \geq W(t_a, t_c) - L$. Since $W(t_a, t_c) = (t_c - t_a)R$, it follows that

$$\begin{aligned} t_c - t_a \leq \frac{V(t_c) - V(t_a^+) + L}{R} &\leq \\ \frac{F_i(t_c) + L - V(t_a^+) + L}{R} = \frac{F_i(t_c) - V(t_a^+) + 2L}{R}. & \end{aligned} \tag{57}$$

To prove the theorem we will find an upper bound to $F_i(t_c)$ and a lower bound to $V(t_a^+)$. Since the approximate virtual start time of the flow increases by σ_i in any time interval $[t_1, t_2]$ during which an amount of bytes $\phi^k \sigma_i$ of the flow are transmitted, and such that there is still backlog at time t_2, it follows that

$$\begin{aligned} F_i(t_c) &= \\ S_i(t_c) + 2\sigma_i &= \\ S_i(t_c) + 2\sigma_i &\leq \\ S_i(t_a^+) + \sigma_i \left\lfloor \frac{Q^k(t_a^+)}{\phi^k \sigma_i} \right\rfloor + 2\sigma_i &\leq \\ S_i(t_a^+) + \sigma_i \frac{Q^k(t_a^+)}{\phi^k \sigma_i} + 2\sigma_i &= \\ S_i(t_a^+) + \frac{Q^k(t_a^+)}{\phi^k} + 2\sigma_i. & \end{aligned} \tag{58}$$

Substituting this inequality and $V(t_a^+) \geq S_i(t_a^+) - \sigma_i$ (derived from (30)) in (57), we get

$$\begin{aligned} t_c - t_a &\leq \\ \frac{S_i(t_a^+) + \frac{Q^k(t_a^+)}{\phi^k} + 2\sigma_i - S_i(t_a^+) + \sigma_i + 2L}{R} &= \\ \frac{\frac{Q^k(t_a^+)}{\phi^k} + 3\sigma_i + 2L}{R}, & \end{aligned} \tag{59}$$

which proves the thesis. □

APPENDIX C

Storage Systems Emulation

WORKING on disk scheduling we had to face the problem of test coverage. In particular, we found quite difficult to predict how changing the physical disks used for testing impacted the performance of scheduling algorithms, or to understand the dependencies between the features of the schedulers and the characteristics of the physical drives. Testing with real hardware would have been the ideal solution, but it requires access to a lot of different, often expensive, media. So we explored the feasibility of using emulation, along with real hardware testing, to better understand the behavior of different scheduling algorithms, and to prevent pitfalls in their deployment.

In this appendix we present one of the results of our effort, GemDisk, a disk emulation tool which we are developing to mimic the behavior of a disk unit (rotational media or even SSD) with configurable characteristics.

C.1. The Problem

Emulation has always been a powerful tool when designing and testing complex systems. Disk devices are a good example of how complex a system can be, and are no exception to that rule. The number of disk drives that have been produced is so high that testing a software solution covering a substantial subset of the existing models is just impossible; so there is the need to isolate their characteristics, and test the behavior of the software systems under exam considering the biggest possible amount of relevant configurations. Doing that using physical systems would still require the access to a big (and growing) number of disk devices, so emulation has to be considered as a cheap, yet accurate, solution to broaden the scope of storage system testing.

The main limit we encountered with the existing emulation/simulation approaches was their level of abstraction; we found either full system emulators, allowing the execution of a full operating system on top of them but not offering an accurate model of disk performance, or disk simulators, very accurate in simulating disk devices, but highly inflexible or limited in the kind of system software they could be used to test.

Given the need for an accurate emulator capable of running real applications, and considered the limitations of the previous solutions, we exploited the flexibility of the FreeBSD GEOM subsystem[1] and the accuracy of DiskSim, a state-of-the-art disk simulator, to create our own emulator. GemDisk is able to run unmodified applications on top of a user-specified mesh of real and emulated devices, using online simulation to drive the response times of the emulated devices.

[1]GEOM is the infrastructure available under FreeBSD to manipulate block device requests; for an introduction to it and for more details please refer to [**Pro09**].

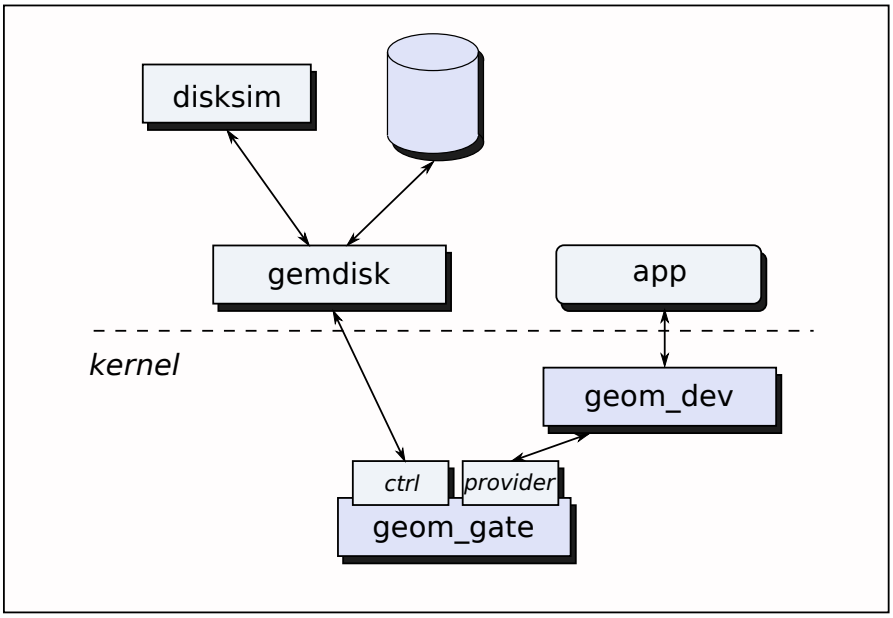


FIGURE 1. Architecture.

C.2. Main Idea

The main idea behind GemDisk is providing virtual devices emulating the behavior of real ones, in a system running on bare metal. To do this we use the GEOM gate class to create a GEOM provider which passes its requests to the userspace and has them served according to an user-level defined policy. The user-level demon controlling both the content and the timing of the requests reinjected back to the kernel uses an internal online simulator to make the provider react like a real device with the configured parameters.

As anticipated, the two dimensions controlled by the emulator are the handling of the backing store for the emulated device, and its temporal behavior. This is reflected by the architecture of our solution, a multithreaded application running in background as a demon, handling the incoming requests (going from the geom gate to userspace) and reinjecting them (from userspace to the geom gate) after a) their content is read/written as requested, and b) the internal simulator says it is the time to do so. We will see how this two-dimensional split can cause problems with requests which the emulated device is ready to complete, but still have their backing store not ready to pass them back to the gate.

Figure 1 shows the architecture of the system. Requests flow from the application to the GEOM device node on top of a GEOM gate provider. The GEOM gate is connected, via a control interface (implemented using character device operations) to the userspace GemDisk demon. The GemDisk demon uses information coming both from actual storage and from a DiskSim instance to answer the requests coming from the gate. Using geom_gate allows us to use only userspace components; we will see that this has its good and bad sides.

C.2.1. GemDisk. GemDisk is a userspace geom gate client, structured as a multithreaded application. Its components are:

(1) the main thread, which initializes the application and then starts executing the DiskSim main loop;
(2) one or more *gatekeepers*, one per gate device, each one listening for the requests coming from the gate geoms, and sending them to the components that process them;
(3) a single (optional) *reaper*, which waits for the completion of the synchronous requests coming from the backing store.

C.2.2. About Time. The single most complex issue to face while implementing GemDisk was timing. An ideal emulator would serve the requests exactly at the time the real hardware would do, so we used real time as a baseline for the emulator. Anyway, in particular when emulating fast drives, the backing store might be slower than the emulated hardware, and it may happen that a request that should have been completed in the emulated system is still under service in the real one.

Please note that most of the benchmarks which act at the block level, reading or writing directly to the device, do not care about the contents of the requests; in this case the user can just specify an empty backing store, bypassing completely the problem, and getting the proper timing from the emulator.

Anyway in the general case this is not true; to "solve" this problem we decided to *slow down* the perceived time for the applications under test. This was done hijacking some of the libc-provided syscalls related to time manipulation for the client processes.

A process issuing synchronous requests only, which is blocked waiting for the completion of the outstanding request, cannot even notice this temporary halt of system time (of course unless it communicates with external applications, with strict clock synchronization requirements). Processes issuing asynchronous requests see a slower system when emulated time is blocked, but generally can still make good use of the timing information they read (being them benchmark applications, they measure time mostly to generate I/O at given rates or to measure latency/throughput of the emulated device, thus they act like they were executed in a system running at emulation time).

C.3. Usage Scenarios

C.3.1. Timing-only Emulation. The first, simple usage of GemDisk is the emulation of the timings of direct disk accesses which do not use the data they read/write. This may seem a useless scenario, in fact a lot of microbenchmarking activity needed to tune a disk scheduler does not use anything it reads or write; of course using a filesystem is not possible.

To support this mode of operation, GemDisk allows the creation of emulated devices with no backing store. The libc hijacking code is not needed because the I/O side of requests is never dispatched, so it can never be late.

C.3.2. In-memory Emulation. If the application under test uses the data it reads or writes, then a backing store must be specified. Anyway using a ramdisk allows bypassing the libc hijacking code, because accessing the backing store is fast (and unless emulating a flash memory it's unlikely to take

more in the real system than in the emulated one). The big limitation of this configuration is that the size of the emulated devices cannot exceed the RAM size available on the system.

C.3.3. Disk-backed Emulation. The complete configuration allows using a file/device as backing store, thus it does not limit the maximum size for the emulated device. In this case time for the clients needs to be virtualized. Here the drawback is that all the applications doing I/O on the emulated devices, and the ones communicating with them need to be synchronized on the same clock driven by the emulator. This prevents the usage of GemDisk in distributed systems.

C.4. Related Work

Storage systems simulation and modeling has a long history, and many of its aspects have been analyzed in the literature. Here we present the background we exploited designing and developing GemDisk.

C.4.1. Simulation and Emulation. The model described in [**RW94**] has been implemented in the simulator described in [**KTR94**]. Anyway the de-facto standard for research works is DiskSim [**BSSG08**], which we describe in more detail later.

The most relevant example of a timing-accurate storage emulator is the Memulator [**GSS+02**], which uses DiskSim, like GemDisk, to control the timing of the emulated device. The key difference between GemDisk and the Memulator is that the former is able to provide accurate simulation results (if the clients used for benchmarking do not use complex time handling techniques—an assumption that proved to be satisfied by many common benchmarking tools) even if the dimension of the emulated device is bigger than system RAM. Note also that RAM can be used as backing storage in GemDisk too, and the geom gate tools shipped with the system can be used to create a setup similar to the remote one implemented in Memulator[2].

Among others, in [**BM95**] the authors use an emulator similar to our one; they use it as a component of their system, consisting in an online filesystem and an offline analysis engine. In [**WAP99**] the authors use an emulation scheme with virtual devices driven by an in-kernel emulator to validate their proposed handling of synchronous writes. Both these approaches are unable to cope with disk sizes exceeding the size of the RAM available in the system.

System level emulators like QEMU [**Bel**] usually do not provide accurate timing for the storage components they emulate, as their focus is mainly in the throughput they can provide. However, more accurate emulators have been proposed, for example SimOS [**RBDH97**] used the simulator described in [**KTR94**] to drive the timings of its storage components.

C.4.2. Parameter Extraction. Even the most accurate model needs an accurate knowledge of its parameters to produce accurate results. Various techniques have been proposed in the past, usually consisting in analyzing the response time of suitably crafted request patterns sent to the disk drive, eventually varying the initial state of the drive itself.

[2]In this setup the emulation is done on a remote machine, connected via a network interface to the system under test; this is useful if the load induced by the emulation component can interfere with the behavior of the application being tested.

The various approaches in literature differ for the methodology of the various solicitations on the drive being analyzed. For example [**TADP00**] uses only microbenchmarks consisting in read and write requests, being portable, but limited in the parameters that can be determined. On the other hand [**WGPW95**] uses SCSI commands to extract more detailed information from the drives. Finally, [**GG99**] presents DIXTrac, a tool for automated parameter extraction, designed to work together with DiskSim, providing it the parameters needed to simulate the analyzed drive.

C.5. Experimental Evaluation

To evaluate the effectiveness of our emulation approach, we tested the accuracy of GemDisk in two different usage scenarios, with two different synthetic workloads injected into the system running the emulator. We focused on accuracy because, by design, GemDisk is able to emulate devices of arbitrary bandwidth requirements, as the "time stealing" mechanism makes the device appear to be faster than it really is according to real time. Apart from that, we used [**GSS+02**] as a model for the evaluation of GemDisk.

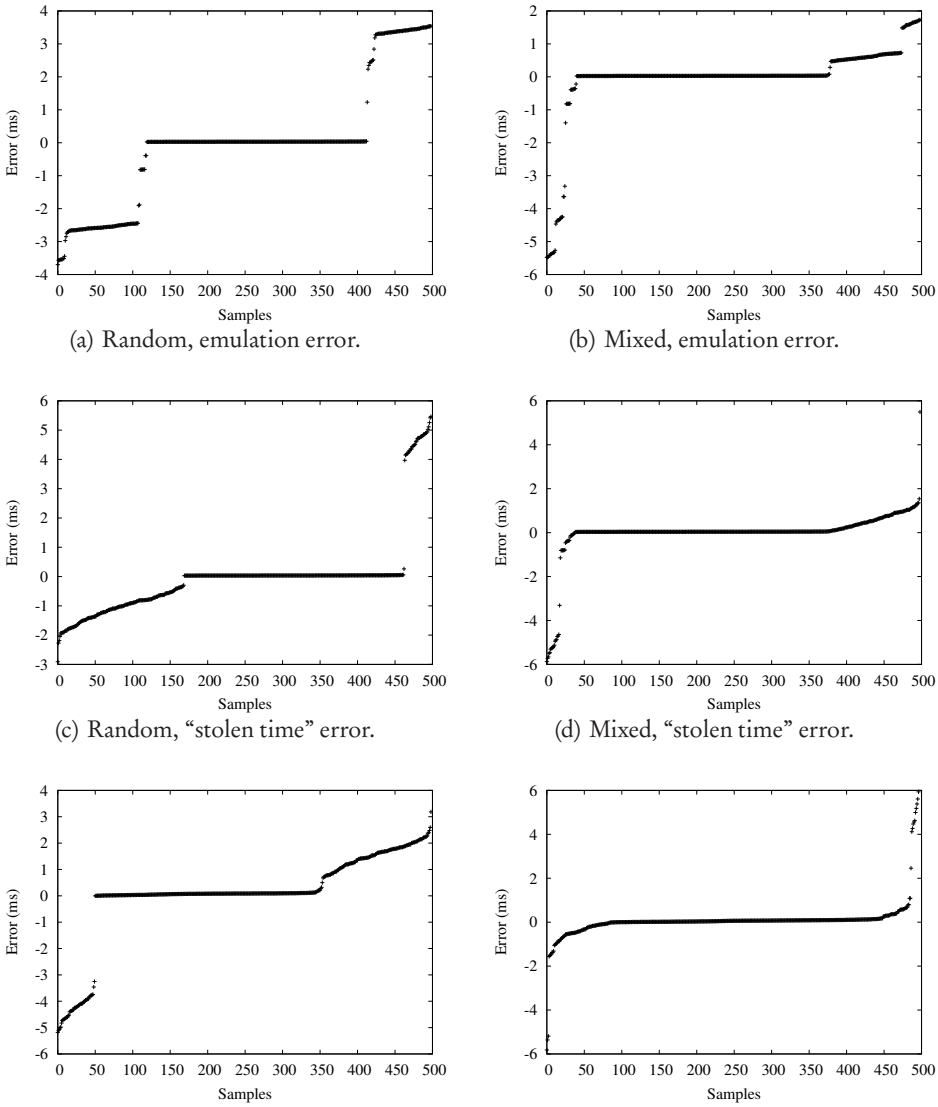
In our experiments we used a low-end setup, composed by an Asus EEEBOX, equipped with an Atom N270 1.60GHz CPU, 1GB RAM and a Western Digital WD3200BEKT hard disk, with 16MB of cache, working at 7200RPM. Since the CPU pressure of the testing application is an issue that must be solved separately (e.g., moving the emulation component to a remote box) we used I/O bound workloads to test the emulator. Our basic aim was at evaluating the timing behavior of the emulated device, comparing it with the reference one provided by a DiskSim simulation of the same workload used for testing.

We generated a couple of different workloads, a *random* one, characterized by zero probability of local or sequential access, and a *mixed* one, with 0.3 probability of local access (i.e., within 500LBNs from the previous request), 0.2 probability of sequential access, and 0.5 probabilty of random access. In both cases the request size was uniformly distributed in [2KiB, 130KiB], the interarrival times were uniformly distributed in [0, 500s], and the number of generated requests was 500.

C.5.1. Emulation Accuracy. The first measure of emulation accuracy we did consisted in comparing the service time of each request in GemDisk with the service time of the same request in the DiskSim simulation, that we called *emulation error*. To do that we developed a simple synthetic trace replay tool that we used to inject requests in the simulator and, in a later step, in the gate device connected to the emulator.

For the simulator we considered the service time as the difference from the simulation time when the request was injected into DiskSim, and its completion time; for GemDisk we considered the difference between when the request reached the emulator and the time when it was inserted back into the gate device.

Figure 2 shows the distributions of the errors obtained during our measurements; in particular Figures 2(a) and 2(b) show the emulation error, in the case of a virtual device not backed by real storage. We can see that most of the samples are around zero, meaning no difference between the completion time in the simulator and the one in the emulator.



(a) Random, emulation error.

(b) Mixed, emulation error.

(c) Random, "stolen time" error.

(d) Mixed, "stolen time" error.

(e) Random, user perceived error.

(f) Mixed, user perceived error.

FIGURE 2. Various error figures obtained with the test workloads.

The figures show that a significative part of the samples completes earlier in the emulator than in the simulator. This is most likely due to the fact that the injection tool in some cases is not able to respect the arrival time of the requests and issues them with some delay, causing a discrepancy between the arrivals seen from the internal simulator with respect to the ones seen in the simulator-only environment. Even if we need to work more on this issue, this behavior seems related more to the way used to inject requests in the system rather than to the emulator design.

On the other hand, errors do not exceed a few ticks, which is compatible with both timer accuracy and preemptions which can cause the GemDisk threads to be descheduled when requests have to be reinjected back to the gate device.

With *"stolen time" error* we indicated the error in the completion time obtained at the reinjection in the gate, but when the time stealing mechanism was active (i.e., with the emulated device backed by real storage). Figures 2(c) and 2(d) show that there is no significative difference with respect to the emulation error with no backing device, indicating that, at least for simple applications the time stealing mechanism is a viable approach. Even if this was not the goal of this tests, the trace contained some amount of parallelism in the requests, and the trace replay tool used asynchronous I/O, so this behavior was not obtained in the most favorable conditions for the mechanism to work.

The last accuracy index we considered was what we called the *user perceived error*, i.e., the difference between the completion times in an emulated system with no backing store and no time stealing, and a system with an emulated drive with backing store and time stealing. Figures 2(e) and 2(f) show that the time stealing mechanism does add some noise, but it remains under the few milliseconds. As we have also seen previously, in our tests there was no substantial accuracy degradation deriving from the use of this mechanism.

C.6. Availability

GemDisk is available at:

`http://feanor.sssup.it/~fabio/freebsd/gemdisk/`

C.7. Future Work

GemDisk is still under development. There is still room for improvements with respect to its accuracy; moreover, we need to test it under more workloads, to better understand its limitations, and we need to extend the time virtualization mechanism to cover more possible users. Another topic we still have to explore is how the same time warping mechanism can be used to provide emulation with multiple instancies of GemDisk and/or on distributed systems.

C.8. Conclusion

We described GemDisk, a GEOM-based timing-accurate disk emulator that can be used to test system performance with a variety of storage system configurations, without the need of buying and assembling the real hardware.

We have also presented what, to the best of our knowledge, is the first attempt to let the storage emulation component drive the time perceived by the applications under test.

Bibliography

[Axb] Jens Axboe. The CFQ I/O scheduler. http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.pdf.

[BBG+99] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *ICMCS '99: Proceedings of the IEEE International Conference on Multimedia Computing and Systems Volume II-Volume 2*, page 400, Washington, DC, USA, 1999. IEEE Computer Society.

[Bel] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46.

[BFQ] `http://algo.ing.unimo.it/people/paolo/disk\_sched.php`.

[BM95] Peter Bosch and Sape J. Mullender. Cut-and-paste file-systems: integrating simulators and file-systems. In *Proceedings of the 1996 USENIX Technical Conference*, pages 307–318. USENIX, 1995.

[BSSG08] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. The DiskSim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University, May 2008.

[BZ96] Jon C. R. Bennett and Hui Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996.

[BZ97] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[Chu01] Guo Chuanxiong. SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks. In *In ACM SIGCOMM 2001*, pages 211–222, 2001.

[Dis] `http://google-opensource.blogspot.com/\2008/08/linux-disk-scheduler-benchmarking.html`.

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM.

[DS94] S. Daigle and J. Strosnider. Disk scheduling for multimedia data streams, 1994.

[GG99] Jiri Schindler Gregory and Gregory R. Ganger. Automated disk drive characterization. Technical report, 1999.

[GMUV07] Ajay Gulati, Arif Merchant, Mustafa Uysal, and Peter J. Varman. Efficient and adaptive proportional share i/o scheduling. Technical report, Hewlett-Packard, November 2007.

[GMV07] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. *SIGMETRICS Perform. Eval. Rev.*, 35(1):13–24, 2007.

[GSS+02] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. Timing-accurate storage emulation. In *In Proceedings of the FAST 2002 Conference on File and Storage Technology)*, pages 75–88. USENIX Association, 2002.

[GVC97] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw.*, 5(5):690–704, 1997.

[ID01] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[IS95] H. Abdel-Wahab I. Stoica. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Technical Report 95-22, *Department of Computer Science, Old Dominion University, November 1995*, Nov 1995.

[JCK04] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 37–48, New York, NY, USA, 2004. ACM.

[Kar] M. Karsten. Approximation of generalized processor sharing with stratified interleaved timer wheels. *Work in progress*.

[Kar06] M. Karsten. SI-WF^{2}Q: WF^{2}Q approximation with small constant execution overhead. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, April 2006.

[KGM95] B. Kao and H. Garcia-Molina. An overview of real-time database systems, 1995.

[KL05] T. Plagemann K. Lund, V. Goebel. APEX: adaptive disk scheduling framework with QoS support. *Multimedia Systems*, 11(1):45–59, 2005.

[KMOR05] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts. Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing. *SIGMETRICS Perform. Eval. Rev.*, 33(1):217–228, 2005.

[KTR94] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical report, Dept. of Computer Science, Dartmouth College, 1994.

[Law07] George Lawton. Powering down the computing infrastructure. *Computer*, 40(2):16–19, 2007.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[LMS04] Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. Tradeoffs between low complexity, low latency, and fairness with deficit round-robin schedulers. *IEEE/ACM Trans. Netw.*, 12(4):681–693, 2004.

[MJR97] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems. guaranteeing timing constraints for disk accesses in rt-mach. *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 155–165, 2-5 Dec 1997.

[Ols05] Robert Olsson. Pktgen: the Linux packet generator. In *Proceedings of the 2005 Ottawa Linux Symposium*, July 2005.

[PG92] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networksthe single node case. In *IEEE INFOCOM '92: Proceedings of the eleventh annual joint conference of the IEEE computer and communications societies on One world through communications (Vol. 2)*, pages 915–924, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Pro09] The FreeBSD Documentation Project. *FreeBSD Handbook*. 2009. `http://www.freebsd.org/doc/en/books/handbook/index.html`.

[qfq] `http://feanor.sssup.it/~fabio/linux/qfq/`.

[RBDH97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7:78–103, 1997.

[RP03] Lars Reuther and Martin Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 374, Washington, DC, USA, 2003. IEEE Computer Society.

[RP06] Sriram Ramabhadran and Joseph Pasquale. The stratified round robin scheduler: design, analysis and implementation. *IEEE/ACM Trans. Netw.*, 14(6):1362–1373, 2006.

[RV04] Luigi Rizzo and Paolo Valente. Hybrid: achieving deterministic fairness and high throughput in disk scheduling. In *Proceedings of CCCT'04*, 2004.

[RW93] A. L. Narasimha Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. In *MULTIMEDIA '93: Proceedings of the first ACM international conference on Multimedia*, pages 225–233, New York, NY, USA, 1993. ACM.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27:17–28, 1994.

[RWW05] A. L. N. Reddy, Jim Wyllie, and K. B. R. Wijayaratne. Disk scheduling in a multimedia I/O system. *ACM Trans. Multimedia Comput. Commun. Appl.*, 1(1):37–59, 2005.

[SBZ99] D.C. Stephens, J.C.R. Bennett, and Hui Zhang. Implementing scheduling algorithms in high-speed networks. *Selected Areas in Communications, IEEE Journal on*, 17(6):1145–1158, Jun 1999.

[SV95] M. Shreedhar and George Varghese. Efficient fair queuing using deficit round robin. In *IEEE/ACM Transactions on Networking*, pages 375–385, 1995.

[SV97] D. Stiliadis and A. Varma. A general methodology for designing efficient traffic scheduling and shaping algorithms. *INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 1:326–335, 1997.

[SV98] Prashant J. Shenoy and Harrick M. Vin. Cello: a disk scheduling framework for next generation operating systems. *SIGMETRICS Perform. Eval. Rev.*, 26(1):44–55, 1998.

[SVC97] Subhash Suri, George Varghese, and Girish Chandranmenon. Leap forward virtual clock: A new fair queuing scheme with guaranteed delays and throughput fairness. In *Proceedings of INFOCOM'97*, 1997.

[SZN00] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Trans. Netw.*, 8(2):185–199, 2000.

[TADP00] N. Talagala, R. Arpaci-Dusseau, and D. Patterson. Micro-benchmark based extraction of local and global disk characteristics. Technical report, Berkeley, CA, USA, 2000.

[Val05] P. Valente. Extending WF^2Q+ to support a dynamic traffic mix. *Advanced Architectures and Algorithms for Internet Delivery and Applications, 2005. AAA-IDEA 2005. First International Workshop on*, pages 26–33, 15-15 June 2005.

[Val07] Paolo Valente. Exact gps simulation and optimal fair scheduling with logarithmic complexity. *IEEE/ACM Trans. Netw.*, 15(6):1454–1466, 2007.

[VLC] `http://www.videolan.org/vlc/`.

[WAEMTG07] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *In Proceedings of the 5th USENIX Conference on File and Storage Technologies. USENIX Association*, 2007.

[WAP99] Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Virtual log based file systems for a programmable disk. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 29–43, Berkeley, CA, USA, 1999. USENIX Association.

[WGP94] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 241–251, New York, NY, USA, 1994. ACM.

[WGPW95] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters, 1995.

[XL02] J. Xu and R. J. Lipton. On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms. In *In Proceedings of ACM SIGCOMM '02*, 2002.