This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

QFQ: Efficient Packet Scheduling With Tight Guarantees

Fabio Checconi, Luigi Rizzo, and Paolo Valente

Abstract—**Packet scheduling, together with classification, is one of the most expensive processing steps in systems providing tight bandwidth and delay guarantees at high packet rates. Schedulers with near-optimal service guarantees and $O(1)$ time complexity have been proposed in the past, using techniques such as *timestamp rounding* and *flow grouping* to keep their execution time small. However, even the two best proposals in this family have a per-packet cost component that is linear either in the number of groups or in the length of the packet being transmitted. Furthermore, no studies are available on the actual execution time of these algorithms. In this paper we make two contributions. First, we present Quick Fair Queueing (QFQ), a new $O(1)$ scheduler that provides near-optimal guarantees and is the first to achieve that goal with a truly constant cost also with respect to the number of groups and the packet length. The QFQ algorithm has no loops and uses very simple instructions and data structures that contribute to its speed of operation. Second, we have developed production-quality implementations of QFQ and of its closest competitors, which we use to present a detailed comparative performance analysis of the various algorithms. Experiments show that QFQ fulfills our expectations, outperforming the other algorithms in the same class. In absolute terms, even on a low-end workstation, QFQ takes about 110 ns for an enqueue()/dequeue() pair (only twice the time of DRR, but with much better service guarantees).**

Index Terms—**Algorithms, communication systems, computer network performance, data structures, packet scheduling.**

I. Introduction

IF AN outgoing link on a network node is fully utilized, the only option to provide bandwidth or delay guarantees on that link is enforcing a suitable packet scheduling policy. Fine-grained per-flow guarantees can be provided with an IntServ [12] approach, but this requires a reservation protocol (with well-known scalability problems) and imposes a nonnegligible load on packet classifiers and schedulers, who have to deal with a potentially large number of flows *in progress* (up to 10^5 according to [10]). Besides memory costs to keep per-flow state, the time complexity and service guarantees of the scheduling algorithm can be a concern. DiffServ [12] "solves" the space and time complexity problem by aggregating flows into a few classes with predefined service levels and schedules the aggregate classes. Per-flow scheduling within each class may still be needed to provide guarantees to individual flows.

The above considerations motivate the interest for packet schedulers that, even in presence of a large number of flows, can offer low complexity and tight guarantees.

Round Robin schedulers have $O(1)$ time complexity, but (with the exception of FRR [24]) have $O(N)$ worst-case deviation with respect to the ideal service that the flow should receive over any given time interval.

More accurate schedulers have been proposed, implementing approximate versions of the worst-case optimal $\mathrm{WF}^2\mathrm{Q}+$ scheduler [2]. Thanks to flow grouping and timestamp rounding, first introduced in [17], they feature $O(1)$ time complexity in the number of flows, and near-optimal deviation from the ideal service (WFI; see Section II). However, even the two most efficient proposals in this class, namely *Group Fair Queueing* (GFQ) [17] and *Simple KPS* (S-KPS) [9], as well as FRR [24], have some nonconstant components in their time complexity, as discussed in Section II, and are significantly slower than Round Robin schedulers. Section VII-C shows some performance comparison.

Our Contributions: In this paper, we first present Quick Fair Queueing (QFQ), a new scheduler with true $O(1)$ time complexity, implementing an approximate version of $\mathrm{WF}^2\mathrm{Q}+$ with near-optimal service guarantees. We then provide an extensive comparison of the actual performance of production-quality implementations of QFQ and of several competing algorithms.

The key contribution of QFQ is the introduction of a novel mechanism (Group Sets, Section IV-3), which removes the linear component (in the number of groups or packet size) from previous quasi-$O(1)$ schedulers. In QFQ, the flow groups are partitioned into four sets, each represented by a machine word and constructed so that all the bookkeeping required to implement scheduling decisions can be done using simple and constant-time CPU instructions such as AND, OR, XOR, and *Find First bit Set*[1] (which we use to implement constant-time searching).

The major improvement of QFQ over the 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. The execution time is within two times that of DRR, and consistently about three times faster than S-KPS, across a wide variety of configurations and CPUs. Speed does not sacrifice service guarantees: The WFI of

Manuscript received November 25, 2010; revised August 26, 2011 and March 10, 2012; accepted July 20, 2012; approved by IEEE/ACM Transactions on Networking Editor I. Keslassy.

F. Checconi was with Scuola Superiore Sant'Anna, Pisa 56127, Italy. He is now with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: fchecco@us.ibm.com).

L. Rizzo is with the Università di Pisa, Pisa 56122, Italy.

P. Valente is with the Università di Modena, Modena 41100, Italy.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2012.2215881

[1]The Find First bit Set instruction (called ffs() or BSR) can locate in constant time the leftmost bit set in a machine word. It uses 1 . . . 3 clock cycles on modern CPUs such as Intel Core 2, Core i7, or Athlon K10.

QFQ is slightly better than S-KPS and close to the theoretical minimum.

Paper Structure: Section II complements this introduction by discussing related work. In Section III, we define the system model and other terms used in the rest of the paper. Section IV presents the QFQ algorithm in detail and illustrates its implementation. The correctness of the properties used in QFQ is then proved in Section V. Section VI gives an analytical evaluation of the (worst-case) service guarantees. In Section VII-A, we present the results of some *ns2* simulations to compare the delay experienced by various traffic patterns under different scheduling policies. Finally, Section VII-B measures the actual performance of the algorithm on a real machine, comparing production-quality implementations of QFQ, S-KPS, and of other schedulers (FIFO, DRR, and WF2Q+).

II. Background and Related Work

Packet schedulers can be classified based on their service properties and time/space complexity. Relevant problem dimensions are the number of flows, N, and the maximum size L of packets in the system. The service metrics defined in the literature try to measure, in various dimensions, the differences between the scheduler under analysis and an *ideal fluid system* that implements perfect bandwidth distribution over any time interval.

Two important service metrics are the *Bit-* and *Time-Worst-case Fair Index* (B-WFI and T-WFI [2], [3]). B-WFI^k (defined in Section VI-A) represents the worst-case deviation, in terms of service, that a flow k may experience *over any time interval* with respect to a perfect weighted bandwidth sharing server during the same interval. T-WFI^k, defined in Section VI-B, expresses similar deviations in terms of time. From the WFIs, it is easy to compute the minimum bandwidth and the worst-case packet completion times guaranteed for a flow. However, the WFIs indicate more than just worst-case packet delays or per-flow *lag* (the difference between the service received in an ideal, perfectly fair system and the one received in the actual system). The WFIs capture the fact that an actual scheduler may serve some packets much earlier than the ideal system, and this may result in long intervals during which a flow receives no service to compensate for that received in advance. This may not affect the lag, but causes extreme burstiness in service, which has bad effects on protocols and applications (e.g., TCP's rate adaptation) as well as on per-flow lag and delay guarantees in a hierarchical setting [2].

In contrast, the WFIs do not measure another important service property: how fairly a scheduler distributes the excess bandwidth when not all the flows are backlogged. This property can be measured with an early metric, called *relative fairness* in [8] and *proportional fairness* in [24], which is equal to the worst-case difference between the *normalized service* (service divided by the flow's weight) given to any two backlogged flows over any time interval [16].

Round Robin Schedulers: Round Robin (RR) schedulers and variants (*Deficit Round Robin* [15]) are the usual choice when fast schedulers are needed. They lend naturally to $O(1)$ implementations with small constants. Several variants have been proposed (*Smoothed Round Robin* [6] and *G-3* [7], *Aliquem* [11], and *Stratified Round Robin* [13]) to mitigate some of their shortcomings (burstiness, etc.). Nevertheless, for all but one of the schedulers in this family, and irrespective of the weight ϕ^k of any flow k, both the flow's packet delay and the B-WFI^k have an $O(NL)$ component.[2]

FRR [24] differs from other RR proposals in that, similarly to QFQ, it divides flows into groups and schedules packets in two phases: First, an extended version of WF^{2}Q [3] schedules groups; following that, an extended version of DRR [15] schedules flows within groups. In FRR, a flow k belongs to group i such that $i = \lceil \log_C \phi^k \rceil$, where ϕ^k is the flow's weight.[3] C is an integer constant that can be freely chosen to set the desired tradeoff between runtime complexity and service guarantees. As shown by its authors in [24, Theorem 4], FRR has T-WFI^k $= 12\frac{CL}{r^k} + (G-1)\frac{L}{R}$, where G is the number of groups, r^k is the minimum bandwidth guaranteed to flow k, and R is the link rate. The time complexity is $O(G \log G)$. T-WFI grows with C, and in the best case ($C = 2$), with weights ranging between 10^{-6} and 1, we would have $G = \lceil \log_C 10^6 \rceil = 20$ and hence T-WFI^k $= 24\frac{L}{r^k} + 19\frac{L}{R}$, much higher than the T-WFI^k of QFQ ($3\frac{L}{r^k} + 2\frac{L}{R}$).

Exact Timestamp-Based Schedulers: To achieve a lower WFI than what is possible with RR schedulers, other, modern scheduler families try to serve flows as close as possible (i.e., not too late and not too early) to the service provided by an internally tracked ideal system, using a concept called *eligibility*. We call them *timestamp-based* schedulers as they typically timestamp packets with some kind of *Virtual Time* function and try to serve them in ascending timestamp order, which has an inherent $\Omega(\log N)$ complexity [23]. This bound is matched by some actual algorithms [21].

With this approach, schedulers such as WF^{2}Q [3] and WF^{2}Q+ [2] offer *optimal* lag, packet delay, and WFI, i.e., they achieve the lowest possible values for a *nonpreemptive* system. In particular, their lag and B-WFI are both $O(L)$ with very small constants (see Section VI-A), much better than the $O(NL)$ of most RR schedulers.

Fast Timestamp-Based Schedulers: Breaking the theoretical $\Omega(\log N)$ bound requires the use of approximate timestamps to reduce the complexity of the sorting steps. Some schedulers (such as GFQ, S-KPS, and LFVC) use this approach to achieve $O(1)$ complexity with respect to the number of flows, while preserving $O(L)$ B-WFI.

GFQ [17] uses variable timestamp rounding, splits flows into G groups, and relies on a calendar queue to sort flows within the same group. Its complexity is $O(G)$. S-KPS [9] uses a data structure called Interleaved Stratified Timer Wheels (ISTW) to execute packet enqueue and dequeue operations at a worst-case cost independent of even the number of groups, though it requires $O(L)$ bookkeeping steps during packet transmissions. Finally, LFVC [20] rounds timestamps to multiples of a fixed constant, relying on van Emde Boas priority queues for sorting (hence $O(\log \log N)$ complexity). Unfortunately, LFVC has a

[2]Such worst case behavior is easy to achieve in practice (e.g., with a few high-weight flows and a large number of low-weight flows). In these circumstances, the high-weight flows will experience a very bursty service, with unpleasant effects for downstream devices and applications.

[3]QFQ also defines groups of flows, but using a different formula, (4).

TABLE I
DEFINITIONS OF THE SYMBOLS USED IN THE PAPER

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 the flow is 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, see Eq. (3)
$V^k(t)$	Virtual time of flow k, see Sec. III-A
S^k, F^k	Virtual start and finish times of flow k, see Eq. (2)
$\hat{S}^k, \hat{F}^k$	Approximated S^k and F^k for flow k, see Section IV-2
i, j	Group index (groups are defined in Sec. IV-1)
S_i, F_i	Virtual start and finish times of group i, see Eq. (6)
σ_i	Slot size of group i (defined in Sec. IV-1, $\sigma_i = 2^i$)
ER, EB, IR, IB	The four sets in which groups are partitioned

worst-case complexity of $O(N)$ because the algorithm maintains separate queues for eligible and ineligible flows, and individual events may require to move up to $O(N)$ flows from one queue to the other.

The use of approximate timestamps has an implication, proved in [23]: Any scheduler based on approximate timestamps has a packet delay with respect to an ideal GPS server larger than $O(L)$. Fortunately, the B-WFI bounds are not affected: GFQ, S-KPS, and our QFQ guarantee the same $O(L)$ B-WFI^k as the optimal schedulers, differing only in the multiplying constant, which is 1 with exact timestamps and slightly larger otherwise (e.g., 3 in the case of QFQ; see Section VI-B). Thus, approximate timestamps still give much better guarantees than RR schedulers.

We should note that the data structures used in the various schedulers differ largely, so that low asymptotic complexity does not necessarily reflect in faster execution times, especially with a small number of flows. Also, there may be dependencies on other parameters, (e.g., GFQ or S-KPS) or worst-case behaviors significantly larger than average (e.g., LFVC).

III. System Model and Definitions

In this section, we give some definitions commonly used in the scheduling literature, and then present the exact WF^{2}Q+ algorithm, which is used as a reference to describe QFQ. For convenience, all symbols used in the paper are listed in Table I. Most quantities are a function of time, but we omit the time argument (t) when not ambiguous and clear from the context.

We consider a system in which N packet flows (defined in whatever meaningful way) share a common transmission link serving one packet at a time. The link has a time-varying rate. 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 assume that $\sum_{k=1}^{N} \phi^k \leq 1$.[4]

A flow is defined as *backlogged/idle* if it owns/does not own packets not yet completely transmitted. We call $B(t)$ the set of flows backlogged at time t. Inside the system, each flow uses a first-in–first-out (FIFO) queue to hold 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.

A. WF^{2}Q+

Here, we outline the original WF^{2}Q+ algorithm for a variable-rate system (see [2] and [18] 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* that 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 nonpreemptive. 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*[5] *in the ideal fluid system*, the next one that would be completed in the fluid system; ties are arbitrarily broken. WF^{2}Q+ is a work-conserving online 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 one or more out-of-order packets have already started.

Virtual Times: The WF^{2}Q+ policy is efficiently implemented by considering, for each flow, a special *flow virtual time* function $V^k(t)$ that grows as the *normalized amount of service* received by the flow (i.e., actual service received, divided by the flow's weight). In addition, when the flow turns from idle to backlogged, $V^k(t)$ is set to the maximum between its current value and the value of a further function, the *system virtual time* $V(t)$, defined below. For each flow k, the value of $V^k(t)$

[4]The implementation of QFQ does not rely on this assumption, and it tracks the actual sum of weights as flows come and go, thus providing tighter guarantees to the backlogged flows.

[5]This property, called "eligibility, " is fundamental in providing small WFI.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

needs to be known (hence computed) only on the following events: 1) when the flow becomes backlogged, or 2) when its head packet completes transmission in the ideal fluid system. The resulting values of $V^k(t)$, called *flow's virtual start* and *finish time*, S^k and F^k, are used to timestamp the flow itself and are computed as

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

where t_p is the time when a packet enqueue/dequeue occurs. $V(t)$ is the *system virtual time* function defined as follows (assuming $\sum \phi^k \leq 1$):

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

Note that $V(t)$ is only computed at discrete times, so the instantaneous link rate does not need to be known, and just $W(t_1, t_2)$ (the amount of data transferred in $[t_1, t_2]$) suffices. At system startup, $V(0) = 0$, $S^k \leftarrow 0$, and $F^k \leftarrow 0$.

Eligibility: Flow k is said to be *eligible* at time t if $V(t) \geq S^k$. This inequality guarantees that the head packet of the flow has already started to be served in the ideal fluid system. Using this definition, $\mathrm{WF^2Q+}$ 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 $\max$ operator in (3) guarantees that the system is work-conserving.

The time complexity in $\mathrm{WF^2Q+}$ comes from three tasks: 1) computing $V(t)$ from (3), which requires to track the minimum S^k and has $O(\log N)$ cost; 2) selecting the next flow to serve among the eligible ones, which requires tracking the minimum F^k among eligible flows, and also has $O(\log N)$ cost at each step; 3) updating eligible flows as $V(t)$ grows. Any change in $V(t)$ can render $O(N)$ flows eligible, and it takes some clever data structure [19] to avoid an $O(N)$ cost.

IV. Quick Fair Queueing

In this section, we describe QFQ. For ease of exposition, the properties requiring long proofs are demonstrated separately in Section V. The scheduler uses the data structure represented in Fig. 1 and relies on three techniques (Flow Grouping, Timestamp Rounding, Group Sets) to perform all computations in $O(1)$ time.

1) Flow Grouping: Each flow k (one of the squares at the bottom of Fig. 1) is statically mapped into one of a finite number of *groups* (the regions at the bottom of the figure). The group i is chosen as

$$i = \left\lceil \log_2 \frac{L^k}{\phi^k} \right\rceil \tag{4}$$

where L^k is the maximum size of the packets for flow k. L^k/ϕ^k is related to the service guarantees given to a flow, so flows with similar guarantees are grouped together.

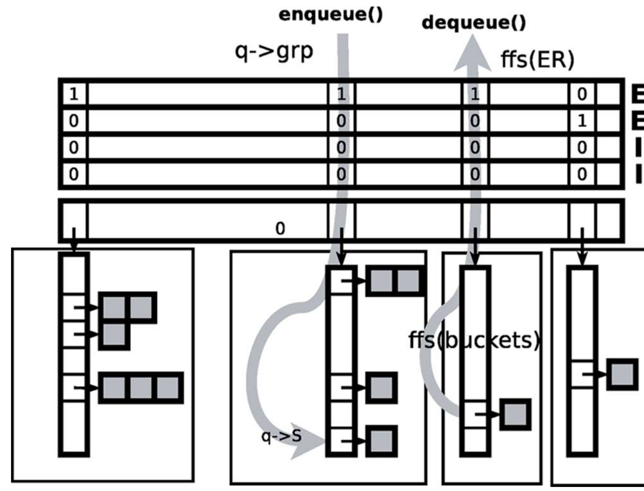

Fig. 1. QFQ at a glance. The figure represents all main data structures used by the algorithm. The four groups sets on the top (see Section IV-3) are stored in a bitmap by index number. The groups (rectangles on the bottom; see Section IV-1) contain the bucket lists and individual flow queues.

In practice, *the number of distinct groups is less than 64* (32 groups suffice in many cases),[6] so a set of groups can be represented by a bitmap in a single machine word.

We define $\sigma_i \equiv 2^i$ (bits) as the *slot size* of the group. Since $\sigma_{i-1} < L^k/\phi^k \leq \sigma_i$, from (2) we have $F^k - S^k \leq \sigma_i$ for any flow k in group i.

2) Timestamp Rounding: Same as other timestamp-based schedulers, QFQ labels flow with both exact and approximate virtual times. The exact values [S^k and F^k, (2)] are used to provide guarantees.[7] The approximate values are defined as

$$\hat{S}^k \leftarrow \left\lfloor \frac{S^k}{\sigma_i} \right\rfloor \sigma_i \quad \hat{F}^k \leftarrow \hat{S}^k + 2\sigma_i \tag{5}$$

(where i is the group index) and are used to compute eligibility and scheduling order. Note that $\hat{S}^k \leq S^k < F^k < \hat{F}^k$, which has useful implications on the runtime and service guarantees of the algorithm.

For each group, QFQ defines

$$S_i = \min_{k \in \mathrm{group}_i} \hat{S}^k \quad F_i = S_i + 2\sigma_i \tag{6}$$

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

QFQ replaces S^k with $\hat{S}^k$ in the definition of the virtual time function

$$V(t_2) \equiv \max\left(V(t_1) + W(t_1,t_2), \min_{k \in B(t_2)} \hat{S}^k \right). \tag{7}$$

It is easy to show that the $\min \hat{S}^k$ in the equation can be calculated as the minimum S_i among the backlogged groups in the system. Same as in (3), at system startup $V(0) = 0$, $S_i \leftarrow 0$, and $F_i \leftarrow 0$.

[6]This is trivially proven by substituting values in (4); as an example, L^k between 64 and 16 kB, ϕ^k between 1 and 10^{-6} yield values between $64 = 2^6$ and $16 \cdot 10^9 \approx 2^{34}$, or 29 groups.

[7]There is one case where the S^k for certain newly backlogged groups can be shifted backwards to preserve the ordering of $\mathbf{EB}$; this exception is described and its correctness is proved in Lemma 4.

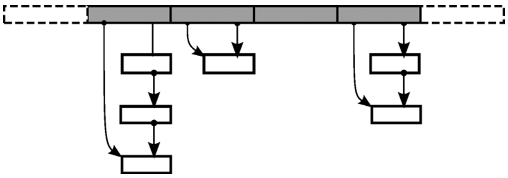

Fig. 2. Representation of bucket lists. The number of buckets (gray, each corresponding to a possible value of $\hat{S}^k$) is fixed and independent of the number of flows in the group.

Finally, note that network interfaces operate on a packet-by-packet basis and do not export a real-time indication of the amount of service provided. To deal with this issue, QFQ computes $V(t)$ based on a simple approximation of the exact amount of service: It assumes that $W(t)$ immediately increments each time a packet is dequeued and updates $V(t)$ accordingly (Section IV–A2).

Timestamp Properties: $\hat{S}^k$ and $\hat{F}^k$ can only assume a limited range of values around $V(t)$ (S^k and F^k have a similar property called Globally Bounded Timestamp or GBT [17]). We can prove (see Theorems 1 and 2 in Section V) that at all times, $\hat{S}^k < V(t) + \sigma_i$.

Furthermore, $\hat{S}^k$ is quantized and can only assume $2+\lceil L/\sigma_i \rceil$ distinct values (remember that $\sigma_i \approx L_k/\phi^k$, so the second term is bounded by the ratio between the min and max packet size in the system). The small number of possible values permits sorting within a group using a constant-time bucket sort algorithm, implemented using a *bucket list* (Fig. 2), a short array with as many buckets as the number of distinct values for $\hat{S}^k$. Each bucket contains a FIFO list of all the flows with the same $\hat{S}^k$ and $\hat{F}^k$. For practical purposes, 64 buckets are largely sufficient, so once again each bucket can be mapped to a bit in a machine word, and a constant-time Find First bit Set instruction can be used to locate the first nonempty bucket, which contains the next flow to serve.

The use of $\hat{S}^k$ in (7) also saves another sorting step because, as we will see, the *group sets* defined in the next section let us compute (7) in constant time.

3) Group Sets and Their Properties: QFQ partitions backlogged groups into four distinct sets (the top rows in Fig. 1). As will be shown in Section IV-A, this reduces scheduling and bookkeeping operations to simple set manipulations, in turn performed with basic CPU instructions such as AND, OR, and FFS on single machine words.

The sets are called $\mathbf{ER}$, $\mathbf{EB}$, $\mathbf{IR}$, and $\mathbf{IB}$ (from the initials of *Eligible, Ineligible, Ready*, and *Blocked*), and the partitioning is done using two properties.

- *Eligible:* Group i is *Eligible* at time t iff $S_i \leq V(t)$. The group is *Ineligible* otherwise.
- *Blocked:* iIndependent of its own eligibility, a group i is *Blocked* if there is some eligible group with higher index and lower finish time. Otherwise, the group is *Ready*.

The "blocked" property is used to partition groups so that within each set $\mathbf{X}$ the group index reflects the ordering by finish time: $(\forall j, i \in \mathbf{X}, j > i \Rightarrow F_j > F_i)$. In particular, the following properties hold.

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's are increasing with i.
3) $\mathbf{ER}$ is sorted by F_i, as proven in Theorem 4 (Section V).
4) $\mathbf{EB}$ is sorted by F_i, as proven in Theorem 5 (Section V).

Hence, the group with the smallest timestamp in a set can be located with an FFS instruction.

Managing Sets: A group can enter any of the four sets when it becomes backlogged or after it is served. After serving a group, QFQ may need to move one or more other groups from one set to another, but only on the paths

$$\mathbf{IB} \rightarrow \mathbf{IR}, \quad \mathbf{IR} \rightarrow \mathbf{ER}, \quad \mathbf{IB} \rightarrow \mathbf{EB}, \quad \mathbf{EB} \rightarrow \mathbf{ER}$$

because the transitions of a group i from ineligible to eligible (driven by the increase of $V(t)$) and from blocked to ready (driven by the increase of the F_j of the group that was blocking group i) are not reversible until group i itself is served.

Moving multiple groups from one set to another requires comparing groups' timestamps with a threshold and affects all groups above or below the threshold. Again, this is done with basic CPU instructions (AND, OR, NOT) *without iterating over the sets* because the ordering by finish time holds for all the four sets, and $\mathbf{IB} \cup \mathbf{IR}$ is sorted by both S_i and F_i. Thus, once the index of the first matching group is known (see Section IV-A.3), all other matching groups are on the same side of the set.

A. Quick Fair Queueing: The Algorithm

We are now ready to describe the details of the QFQ algorithm.

1) Packet Enqueue: Function *enqueue()*, shown in Fig. 3, is called on the arrival of a packet.

As a first step, the packet is appended to the flow's queue, and nothing else needs to be done if the flow is already backlogged. Otherwise, the flow's timestamps are updated (lines 7 and 8), and line 10 checks whether the group's state needs to be updated. This happens if the group was idle, or if the new flow causes the group's timestamp to decrease (in this case, the group was ineligible: line 7 implies $V(t) \leq S^k$, the second test in line 10 succeeds if $S^k < S_i$, hence $V(t) < S_i$). The update is done by lines 11–17, which possibly remove the group from the ineligible sets (lines 13 and 14) and update the group's timestamps (being the slot size 2^i, the start time calculation only needs to clear the last i bits of S_k, line 15).

Once the group's timestamps are set, a constant-time bucket insert (line 18) sorts the flow with respect to the other flows in the group. At this point, if needed, $V(t)$ is updated according to (2). Finally, function *compute_group_state()* (Section IV-A.3/Fig. 5) computes the new state of the group (which may have changed because of the new values of S_i, F_i, and $V(t)$), and line 27 puts the group in its new set.

Note that an enqueue does not move other groups across sets: Their eligibility remains the same ($V(t)$ changes only if all other

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.


```
// Enqueue the input pkt of the input flow
enqueue(in: pkt, in: flow) {
  append(pkt, flow.queue); // always enqueue
  if (flow.queue.head != pkt)
      return; // Flow already backlogged, we are done.
  // Update flow S^k and F^k according to Eq. (2).
  flow.S = max(flow.F, V) ;
  flow.F = flow.S + pkt.length/flow.weight;
  g = flow.group; // g is our group
  if (g.bucketlist.headflow == NULL || flow.S < g.S) {
      // Group g is surely idle or not eligible. Remove from IR and IB if
      // there, and compute the new group S_i and F_i from Eq. (5) and (6).
      set[IR] &= ~(1 << g.index);
      set[IB] &= ~(1 << g.index);
      g.S = flow.S & ~(g.slot_size - 1);
      g.F = g.S + 2*g.slot_size;
  }
  bucket_insert(flow, g);

  // If there is some backlogged group, at least one is
  // in ER; otherwise, make sure V >= g.S
  if (set[ER] == 0 && V < g.S)
      V = g.S;

  // compute new state for g, and insert in the proper set
  state = compute_group_state(g) ;
  set[state] |= 1 << g.index;
}

// Compute the group's state, see Section IV-3
int compute_group_state(in: g) {
  // First, eligibility test
  s = (g.S <= V) ? ELIGIBLE : INELIGIBLE;
  // Find lowest order group x > group.index. This is the group that
  // might block us. ffs_from(d, i) returns the index of the first bit set
  // in d after position i
  x = ffs_from(set[ER], g.index);
  s += (x != NO_GROUP && groups[x].F < g.F) ?
        BLOCKED : READY;
  return s;
}
```

Fig. 3. *enqueue()* function, called on packet arrivals, and *compute_group_state()* that implements the tests for eligibility and readiness.

```
1  packet dequeue() { // Return the next packet to serve
2    if (set[ER] == 0)
3        return NULL;
4    // Dequeue the first packet of the first flow of the group
5    // in ER with the smallest index
6    g = groups[ffs(set[ER])];
7    flow = bucket_head_remove(g.bucketlist);
8    pkt = head_remove(flow.queue);
9
10   // Update flow timestamps according to Eq. (2)
11   flow.S = flow.F ;
12   p = flow.queue.head; // next packet in the queue
13   if (p != NULL) {
14       flow.F = flow.S + p.length/flow.weight;
15       bucket_insert(flow, g) ;
16   }
17
18   old_V = V; // Need the old value in make_eligible()
19   V += pkt.length; // Account for packet just served
20
21   old_F = g.F ; // Save for later use
22   if (g.bucketlist.headflow == NULL) {
23       state = IDLE ; // F not significant now
24   } else {
25       g.S = g.bucketlist.headflow.S ;
26       g.F = g.bucketlist.headflow.F ;
27       state = compute_group_state(g) ;
28   }
29
30   // If g becomes IDLE, or if F has grown, may need to
31   // unblock other groups and move g to its new set
32   if (state == IDLE || g.F > old_F) {
33       set[ER]    &= ~(1 << g.index);
34       set[state] |=   1 << g.index ;
35       unblock_groups(g.index, old_F);
36   }
37
38   x = set[IR] | set[IB]; // all ineligible groups
39   if (x != 0) { // Someone is ineligible, may need to
40                 // bump V up according to Eq. (3)
41       if (set[ER] == 0)
42           V = max(V, groups[ffs(x)].S) ;
43       // Move newly eligible groups from IR/IB to ER/EB
44       make_eligible(old_V, V) ;
45   }
46   return pkt ;
47 }
```

Fig. 4. *dequeue()* function, described in Section IV-A.2.

groups are idle); blocked/ready states also remain unchanged, though for less intuitive reasons.[8]

2) Packet Dequeue: Function *dequeue()* in Fig. 4 is called to return the next packet to send.

The packet selection (lines 2–8) is straightforward. If there are queued flows, at least one flow is eligible, so **ER** is not empty: A first FFS instruction (line 6) picks the group with the lowest index in **ER**, then another FFS is used to locate the first flow in the bucket list (line 7), and the head packet from that flow is the next packet to serve.

Before returning, the function updates the scheduler's data structures in preparation for further work. The flow's timestamps are updated, and the flow is possibly reinserted in the bucket list (lines 10–16). Virtual time is increased in line 19 to reflect the service of the packet selected for transmission. Next (lines 21–28), the group's timestamps and state are updated. If the group has increased its finish time or it has become idle (lines 32–36), it is moved to the new set, and function *unblock_groups()* described in Section IV-A.3 possibly unblocks other groups.

Finally, lines 38–45 make sure that at least one backlogged group is eligible by bumping up V if necessary and moving groups between sets using function *make_eligible()*, which will be discussed next.

3) Support Functions: Fig. 5 documents the remaining support functions, mostly used in the *dequeue()* code.

Function *move_groups()* uses simple bit operations to move groups with indexes in *mask* from set *src* to set *dest*.

Function *make_eligible()* determines which groups become eligible as $V(t)$ grows after serving a flow. The properties of rounded timestamps are used to implement the check in constant time. The algorithm is explained with the help of Fig. 6, which gives a graphical representation of the possible values of S_i's and $V(t)$, and the binary representations of $V(t)$ (the vertical strings of binary digits). Since slot sizes are powers of two ($\sigma_i = 2^i$), the binary representation of the timestamps of the ith group's ends with $i-1$ zeros; in any given slot belonging to group i, the value of the ith bit is constant during the whole slot. Whenever the ith bit of $V(t)$ changes, the virtual time enters a

[8] If the new packet belongs to an already backlogged flow, its group does not change its finish time, so the readiness of others cannot be affected. Otherwise, the group j containing flow k just became backlogged, or its finish time decreased. However, $S^k \geq V(t)$, hence $F_j > \lfloor V(t)/\sigma_j \rfloor \sigma_j + 2\sigma_j$. 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, $j > i \implies \sigma_j \geq 2\sigma_i$, so $F_j \geq F_i$ and the newly backlogged group j cannot block a previously Ready group, even in the worst case (largest possible F_i, smallest possible F_j).

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.


```
1  // Move the groups in mask from the src to the dst set
2  move_groups(in: mask, in: src, in: dest) {
3     set[dest] |= (set[src] & mask) ;
4     set[src]  &= ~(set[src] & mask) ;
5  }
6
7  // Move from IR/IB to ER/EB all groups that become eligible as V(t)
8  // grows from V1 to V2. This uses the logic described in Fig. 6
9  make_eligible(in: V1, in: V2) {
10    // compute the highest bit changed in V(t) using XOR
11    i = fls(V1 ^ V2);
12    // mask contains all groups with index j <= i
13    mask = (1 << (i+1)) - 1 ;
14    move_groups(mask, IR, ER) ;
15    move_groups(mask, IB, EB) ;
16 }
17
18 // Unblock groups after serving group i with F=old_F
19 unblock_groups(in: i, in: old_F) {
20    x = ffs(set[ER])] ;
21    if (x == NO_GROUP || groups[x].F > old_F) {
22          // Unblock all the lower order groups (Theorem 6)
23          // mask contains all groups with index j < i
24          mask = (1 << i) - 1 ;
25          move_groups(mask, EB, ER) ;
26          move_groups(mask, IB, IR) ;
27    }
28 }
```

Fig. 5. Functions to manage group sets after a flow has been served (see Section IV-A.2).

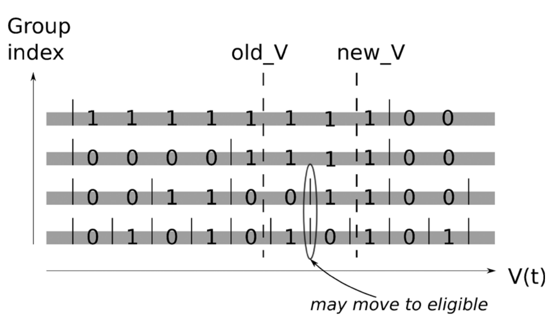

Fig. 6. Tracking group eligibility. On the transition of V from old_V to new_V; the highest bit that flips across the transition is the second one, so groups zero and one are the candidates to become eligible (see Section IV-A.3).

new slot of size 2^i. As a consequence, on each $V(t)$ update, the highest bit j that changes in $V(t)$ indicates that all backlogged groups $G_i, i \leq j$ are now eligible.

This is exactly the algorithm implemented by function *make_eligible()*: It computes the index j using a XOR followed by a *Find Last Set* (FLS) operation, then computes the binary mask of all indexes $i \leq j$ and calls function *move_groups* to move groups whose index is in the mask from $\mathbf{IR}$ to $\mathbf{ER}$ and from $\mathbf{IB}$ to $\mathbf{EB}$.

Function *unblock_groups()* updates the set of blocked groups. When the group j under service increases its finish time or becomes idle, some groups $i < j$ blocked by j might become ready, i.e., do not violate anymore the ordering of $\mathbf{ER} \cup \mathbf{IR}$. Theorem 6 in Section V proves which groups can be unblocked; line 21 verifies the preconditions, and lines 25–26 move groups to $\mathbf{ER}$ and $\mathbf{IR}$ as needed.

B. Time and Space Complexity

From the listings, it is clear that QFQ has $O(1)$ time complexity on packet arrivals and departures: The algorithm has no loops, and 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 (for efficiency, divisions are implemented as multiplications by the inverse of the weight).

Note that while the algorithm has been described assuming $\phi = \sum \phi_k \leq 1$, the actual implementation supports arbitrary values for ϕ, replacing line 19 of the dequeue function with $V+ = \mathrm{pkt.length}/\phi$. Changes in ϕ are tracked in real time as flows come and go: New flows increase ϕ immediately (this does not violate guarantees), while dead flows are expired lazily, using the technique shown in [22].

In terms of space, the per-flow overhead is approximately 24 B (two timestamps, weight, group index, and one pointer). Each group contains a variable number of buckets (32 in the worst case, requiring one pointer each), plus two timestamps and a bitmap. Finally, the main data structure contains five bitmaps, the sum of weights, and a timestamp. Overall, even a large configuration will require 4 kB of memory to hold the entire state of the scheduler.

QFQ has very good memory locality. On each *enqueue()* or *dequeue()* request, the algorithm only touches the internal memory (the 4 kB mentioned above) and the descriptor of the single flow involved in the operation. This is very beneficial both for software and hardware implementations.

V. Proofs of the Properties Used in QFQ

In this section, we prove the properties used in Section IV. The proofs are complete, but slightly condensed due to space constraints. All symbols are defined in Table I, and quantities $(V(t), S^k(t), \ldots)$ are computed as described in the QFQ algorithm.

Hereafter, we explicitly indicate the time at which any timestamp is computed to avoid ambiguity. Given a generic function of time $f(t)$, we define $f(t_1^+) \equiv \lim_{t \to t_1^+} f(t)$. 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.

We define the following two notations for convenience:

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

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

$$\lfloor y \rfloor_{\sigma_i} \leq \lceil x \rceil_{\sigma_i}. \tag{8}$$

In fact, x can be written as $x = n\sigma_i + \delta$, with $0 \leq \delta < \sigma_i$. If $\delta = 0$, then $y < (n+1)\sigma_1 \Rightarrow \lfloor y \rfloor_{\sigma_i} \leq n\sigma_i$, $\lceil x \rceil_{\sigma_i} = n\sigma_i$, 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 as well.

A. Group GBT Under QFQ

We start by proving per-group upper bounds for $S_i(t) - V(t)$ (Theorem 1) and for $V(t) - F_i(t)$ (Theorem 2, supported by the two long Lemmas 1 and 2). The two bounds represent a group-based variant of the GBT property, normally defined for the flow timestamps in an exact virtual-time-based scheduler. We will use these bounds to prove both the properties of the data structure and the B/T-WFI of QFQ. Lemmas 1 and 2 are adapted versions of the ones in [9], repeated here for convenience and with permission from the author.

Theorem 1: Upper bound for $S_i(t) - V(t)$. For any backlogged group i and $\forall t$

$$S_i(t) \le \left\lceil \frac{V(t)}{\sigma_i} \right\rceil \sigma_i = \lceil V(t) \rceil_{\sigma_i}. \tag{9}$$

Proof: By definition (5), 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, $S_i(t) \le \hat{S}^k(t) = \lfloor S^k(t) \rfloor_{\sigma_i}$. It follows that, if $S^k(t) < V(t) + 2\sigma_i$, then (9) trivially holds. Hence, to prove (9), we actually prove the latter inequality, 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 $S^k(t)$ 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 $t_{\mathrm{p}} < t_1$.

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 $F^k(t_1) = S^k(t_{\mathrm{p}}^+) = S^k(t_{\mathrm{p}}) + l^k(t_{\mathrm{p}})/\phi^k \le V(t_{\mathrm{p}}) + \sigma_i + l^k(t_{\mathrm{p}})/\phi^k \le V(t_{\mathrm{p}}) + 2\sigma_i < V(t_{\mathrm{p}}^+) + 2\sigma_i$, which proves the thesis. ■

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

$$\sum_{k \in I(t)} \left(l^k(t) + \phi^k \left[V' - F^k(t) \right] \right) \le V' - V(t) \tag{10}$$

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 $l^k(t) \le \phi^k[F^k(t) - V(t)]$. Therefore, $0 \ge \sum_{k \in I(t)} \{l^k(t) + \phi^k[V(t) - F^k(t)]\} = \sum_{k \in I(t)} \{l^k(t) + \phi^k[V(t) - V'] + \phi^k[V' - F^k(t)]\}$. This implies $\sum_{k \in I(t)} \{l^k(t) + \phi^k[V' - F^k(t)]\} \le \sum_{k \in I(t)} \phi^k[V' - V(t)] \le V' - V(t)$, where the last passage uses $\sum_{k \in I} \phi^k \le 1$. ■

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

$$\sum_{k \in X(t, V')} \left(l^k(t) + \phi^k \left[V' - F^k(t) \right] \right) \le L + V' - V(t). \tag{11}$$

Proof: The proof is by induction over those events that change the terms in (11): *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 (11) 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, which 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 subcases. 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. In both subcases, 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 (11) remains unchanged and the lemma holds.

Virtual Time Jump: After a virtual time jump, all backlogged flows have $S^k(t_1^+) \ge \hat{S}^k(t_1^+) \ge V(t_1^+)$. With regard to the 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 (11) decrease. In the end, $S^k(t_1^+) \ge V(t_1^+)$ for all flows, and if $V' \ge V(t_1^+)$, then Lemma 1 applies and the lemma holds. For other V' in $[V(t_1^+) - L, V(t_1^+)]$, the additional L term in (11) 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 $\hat{F}^k(t_1)$.

Case 1) $V' \ge \hat{F}^k(t_1)$. $V(t)$ is incremented, and hence the right side of (11) 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 for this case.

Case 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, $V(t_1) < \hat{S}^h(t_1) \le S^h(t_1)$ for all flows in $X(t_1, V')$. Lemma 1 applies then for all $V' \ge V(t_1)$, i.e.,

$$\sum_{k \in X(t_1^+, V')} \left(l^k(t_1) + \phi^k \left[V' - F^k(t_1) \right] \right) \le V' - V(t_1). \tag{12}$$

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

$$\begin{aligned} V' - V(t_1) &\le (L - l^k) + V' - (V(t_1^+) - l^k) \\ &= L + V' - V(t_1^+) \end{aligned} \tag{13}$$

and the lemma holds after service also in this case, thus completing the proof. ■

Theorem 2: Upper bound for $V(t) - F_i(t)$ For any backlogged group i

$$V(t) \le F_i(t) + L. \tag{14}$$

Proof: To prove the thesis, we will actually prove, by contradiction, the more general inequality $V(t) \le \hat{F}^k(t) + L$ for a generic flow k of group i. The only event that could lead to a violation of the assumption is serving a packet. Assume that at $t = t_1 : V(t_1) = V_1$ 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.

Case 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 2 with $t = t_1$ and $V' = \hat{F}_2$ results in

$$\sum_{k \in X(t_1, \hat{F}_2)} l^k(t_1) + \sum_{k \in X(t_1, \hat{F}_2)} \left(\hat{F}_2 - F^k(t_1)\right) \phi^k \leq L + \hat{F}_2 - V(t_1). \quad (15)$$

Because $F^k(t) \leq \hat{F}^k(t)$, the second term on the left side of the inequality is nonnegative, and therefore $V_1 + l_p \leq V_1 + \sum_{k:\hat{F}^k(t) \leq \hat{F}_2} l^k(t) \leq V_1 + L + \hat{F}_2 - V_1 \leq \hat{F}_2 + L$.

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

Case 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, $\hat{F}_2 \geq \hat{S}_2 \geq V_2 \geq V_2 - L$. This concludes the proof. ■

B. Proofs of the Data Structure Properties

We can now prove the ordering properties of group sets, considering the two events that can change the set membership: packet enqueue and packet dequeue. We start by proving the following theorem, which bounds the number of possible timestamps within a group.

Theorem 3: At all times, only the first $2 + \lceil \frac{L}{\sigma_i} \rceil$ 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 Theorem 1, from (2) $S^k(t_\mathrm{p}^+)$ may be equal to either: 1) $V(t_\mathrm{p})$, or 2) $F^k(t_\mathrm{p})$, where we assume $F^k(t_\mathrm{p}) = 0$ if $p^{k,l}$ is the first packet of the flow to be enqueued/dequeued.

In the first case, according to (14), $S^k(t_\mathrm{p}^+) = V(t_\mathrm{p}) \leq F_i(t_\mathrm{p}) + L \leq S_i(t_\mathrm{p}) + 2\sigma_i + L \leq S_i(t_\mathrm{p}) + 2\sigma_i + \lceil \frac{L}{\sigma_i} \rceil \sigma_i$. In the second case, neglecting the trivial subcase $F^k(s^{k,l-1+}) = 0$, we can consider that flow k had to be a head flow when $p^{k,l-1}$ was served. Hence, according to (5), $S^k(t_\mathrm{p}) < S_i(t_\mathrm{p}) + \sigma_i$. From (2), this implies $S^k(t_\mathrm{p}^+) = F^k(t_\mathrm{p}) < S_i(t_\mathrm{p}) + 2\sigma_i$.

Considering both cases, it follows that, $\forall t\ S^k(t) - S_i(t) < (2 + \lceil \frac{L}{\sigma_i} \rceil)\sigma_i$, which proves the thesis. ■

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

Lemma 3: Let $\bar{t}$ 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 $F_h(\bar{t}) \leq F_i(\bar{t}^+)$ 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 $S_i(\bar{t}^+) \geq \lfloor V(\bar{t}) \rfloor_{\sigma_i}$ both if the group becomes backlogged and if $F_i(t)$ decreases. Substituting this inequality, which finally holds also if the group becomes eligible at time $\bar{t}$, and (9) in the following difference we get:

$$\begin{aligned} F_h(\bar{t}) - F_i(\bar{t}^+) &= S_h(\bar{t}) + 2\sigma_h - S_i(\bar{t}^+) - 2\sigma_i \\ &\leq \lceil V(\bar{t}) \rceil_{\sigma_h} - \lfloor V(\bar{t}) \rfloor_{\sigma_i} + 2\sigma_h - 2\sigma_i \\ &\leq \lceil V(\bar{t}) \rceil_{\sigma_i} - \lfloor V(\bar{t}) \rfloor_{\sigma_i} + 2\sigma_h - 2\sigma_i \\ &\leq 0 \end{aligned} \quad (16)$$

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

The following theorem guarantees that **ER** is always ordered by virtual finish times.

Theorem 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 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 IV-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 anymore, $F_i(t_1)$ 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 by QFQ when inserting a newly backlogged group into **EB**. Hence, this is the right place to explain in detail this operation.

When an idle group i becomes blocked after enqueueing a packet of a flow k at time t_p, the timestamps of flow k are not updated using the following variant of (2):

$$\begin{aligned} S^k(t_\mathrm{p}^+) &\leftarrow \max\left[\min(V(t_\mathrm{p}), F_b(t_\mathrm{p})),\ F^k(t_\mathrm{p})\right] \\ F^k(t_\mathrm{p}^+) &\leftarrow S^k + l^k(t_\mathrm{p}^+)/\phi^k \end{aligned} \quad (17)$$

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

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

With regard to worst-case service guarantees, in case $V(t_{\mathrm{p}}) > F_b(t_{\mathrm{p}})$ in (17), 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 worst-case schedule for all the flows is the same as if the packet had actually arrived at a time instant such that the would have got exactly those timestamps without using any back-shifting. This is what happens using (17), for the following reason. Should the packet that lets group i become backlogged have arrived at a time instant $\bar{t}_{\mathrm{p}} \le t_{\mathrm{p}}$ at which $V(\bar{t}_{\mathrm{p}}) = F_b(t_{\mathrm{p}})$, group i would have however got a virtual finish time higher than $F_b(t_{\mathrm{p}})$. Hence, group i should not have been served before group b, exactly as it happens in the schedule resulting from timestamping group i with (17) at time t_{p}.

We can now introduce the intermediate lemma we need to finally prove the ordering in $\mathbf{EB}$.

Lemma 4: If a pair of groups h and i with $h < i$ are blocked at a generic time instant t_2, then $S_h(t_2) \le F_i(t_2)$.

Proof: We consider two alternative cases. The first is that $S_h(t_2)$ has been last updated at a time instant $t_1 \le t_2$ using (17). The second is that, according to (2) and (5), there are at least one head flow k of group h and a time instant $t_1 \le t_2$ such that $S_h(t_2) = \lfloor F^k(t_1) \rfloor_{\sigma_h}$.

In the first case, we have $S_h(t_2) \le F_b(t_1)$, where b is the lowest-order group in $\mathbf{ER}$ such that $b > h$. We can consider two subcases. First, group i is already backlogged and eligible at time t_1. It follows that, if $i \ge b$, then $F_i(t_1) \ge F_b(t_1)$. 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, $F_i(t_2) \ge F_i(t_1) > F_b(t_1) = S_h(t_2)$, and the thesis holds. In the other subcase, i.e., group i is not ready and eligible at time t_1, thanks to Lemma 3, group i cannot happen to have a virtual finish time lower than $F_h(t_1)$ during $(t_1, t_2]$. Hence, $F_i(t_2) \ge F_h(t_1) = F_h(t_2) > S_h(t_2)$.

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 $t_0 \le t_1$. 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

$$F_h(t_0) \ge F^k(t_0) > F_i(t_2). \tag{18}$$

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. It follows that $F_i(t_0) \ge F_h(t_0)$ should hold. This inequality and (18) would imply $F_i(t_0) > F_i(t_2)$. Should $F_i(t)$ not decrease during $[t_0, t_2]$, the absurd $F_i(t_2) > F_i(t_2)$ would follow. However, 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 $\forall t \in [t_0, t_2]$ $V(t) \ge S_h(t_0)$, according to (2), (5), and (18), $F_{i,\min} \ge \lfloor S_h(t_0) \rfloor_{\sigma_i} + 2\sigma_i \ge S_h(t_0) - \sigma_h + 2\sigma_i = F_h(t_0) - 3\sigma_h + 2\sigma_i > F_h(t_0) > F_i(t_2)$, 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, $S_h(t_2) \le F_i(t_2)$ must hold. ■

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

Theorem 5: Set $\mathbf{EB}$ 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 ordering of $\mathbf{EB}$ 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 $\mathbf{IB}$ to $\mathbf{EB}$.

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 $\mathbf{EB}$ is ordered up to time t_1. Thanks to Lemma 3, $F_i(t_1) \le F_j(t_1^+)$, and hence the event cannot let group i become blocked.

Suppose now that, at time t_1, group i enters $\mathbf{EB}$ 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 any two blocked groups $h < i$ and $j > i$, $F_h(t_1) \le F_i(t_1^+)$ and $F_i(t_1^+) \le F_j(t_1)$ hold (where $F_i(t_1^+) = F_i(t_1)$ in case group i enters $\mathbf{EB}$ from $\mathbf{IB}$).

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

$$\begin{aligned} &F_i(t_1^+) - F_h(t_1) \ge \\ &F_i(t_1) + \sigma_i - S_h(t_1) - 2\sigma_h \ge 0 \end{aligned} \tag{19}$$

where the last inequality follows from $\sigma_i \ge 2\sigma_h$. On the other hand, if group i enters $\mathbf{EB}$ from $\mathbf{IB}$, then $S_i(t_1) \ge S_h(t_1)$ because either group h was already eligible before time t_1, or group h belonged to $\mathbf{IB}$, which is ordered by virtual start time according to Section IV-3, item 2. This implies $F_i(t_1) \ge F_h(t_1)$ because $\sigma_i \ge 2\sigma_h$.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

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

$$S_i(\bar{t}^+) \le F_b(\bar{t}) \le F_j(\bar{t}) - \sigma_j \tag{20}$$

where the last inequality follows from $F_b(\bar{t}) < F_j(\bar{t})$ and the fact that both $F_j(\bar{t})$ and $F_b(\bar{t})$ are integer multiples of σ_j. Using (20), we have $F_i(\bar{t}^+) = S_i(\bar{t}^+) + 2\sigma_i \le F_j(\bar{t}) - \sigma_j + 2\sigma_i$, i.e.,

$$F_i(\bar{t}^+) \le F_j(\bar{t}) \tag{21}$$

because, since $j > i$, $\sigma_j \ge 2\sigma_i$. ■

Finally, we can prove the theorem that allows QFQ to quickly choose the groups to move from $\mathbf{EB}/\mathbf{IB}$ to $\mathbf{ER}/\mathbf{IR}$.

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

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 $F_i(\bar{t}) < F_j(\bar{t})$ must hold for group i not to be blocked, and both $F_i(\bar{t})$ and $F_j(\bar{t})$ are integer multiples of σ_i, then $F_i(\bar{t}) \le F_j(\bar{t}) - \sigma_i$. Combining this inequality with Lemma 4, we get $S_h(\bar{t}) \le F_j(\bar{t}) - \sigma_i$, and hence, considering that $\sigma_i \ge 2\sigma_h$, $F_h(\bar{t}) = S_h(\bar{t}) + 2\sigma_h \le F_j(\bar{t}) - \sigma_i + 2\sigma_h \le F_j(\bar{t})$. This contradicts the fact that group j blocks group h.

As a consequence, if $F_i(t)$ increases, then, thanks to (19) and (21), 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. ■

VI. Service Guarantees

Service guarantees are an important parameter of any scheduling algorithm. In this section, we compute various service metrics for QFQ: in particular, we will derive two bit guarantees—the B-WFI and *relative fairness*—and one time guarantee—the T-WFI.

A. Bit Guarantees

The B–WFI^k guaranteed to a flow k is defined as[9]

$$\text{B-WFI}^k \equiv \max_{[t_1,t_2]} \left(\phi^k W(t_1,\, t_2) - W^k(t_1,\, t_2)\right) \tag{22}$$

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.

Theorem 7: B-WFI for QFQ. For a flow k belonging to group i, QFQ guarantees

$$\text{B-WFI}^k = 3\phi^k\sigma_i + 2\phi^k L + L^k. \tag{23}$$

[9]This definition is slightly more general than the original one in [2], where t_2 was constrained to the completion time of a packet.

IMPORTANT NOTE: Flow k belongs to group i, so $\phi^k\sigma_i$ varies between L^k and $2L^k$ and the B-WFI^k is always bounded by a small multiple of the packet size, same as for other near-optimal schedulers. In addition, Theorem 1, as well as the theorems and lemmas on which it depends, are proven without ever using the link rate. Hence, *this theorem holds also for time-varying link rates*.

Proof: In this proof, we express timestamps ($V(t)$, $F^k(t)$, etc.) as functions of time to avoid ambiguities. We consider two cases. The first one is when flow k is eligible at time t_1. In this case, as for the virtual time $V^k(t)$ of flow k in the real system, consider that $V^k(t_1) \le F^k(t_1)$, and $V^k(t_2) \ge S^k(t_2)$ trivially hold. In addition, $\forall t, V(t) \le F_i(t) + L$ as proven in Theorem 2, then $S_i(t_2) = F_i(t_2) - 2\sigma_i > V(t_2) - L - 2\sigma_i$. Hence, we have

$$\begin{aligned} V^k(t_1,t_2) &\ge S^k(t_2) - F^k(t_1) \\ &> S_i(t_2) - (S^k(t_1) + \sigma_i) \\ &> V(t_2) - L - 2\sigma_i - (V(t_1) + \sigma_i) \\ &= (V(t_2) - V(t_1)) - L - 3\sigma_i \\ &> W(t_1,t_2) - 2L - 3\sigma_i. \end{aligned} \tag{24}$$

The last inequality follows from the fact that, because of the immediate increment of $V(t)$ as a packet is dequeued (see *updateV()*), $V(t_2) - V(t_1) \ge W(t_1,t_2) - L$. To complete the proof for this case, consider that, for $V^k(t)$ to comply with the immediate increment of $V(t)$, also $V^k(t)$ must increment immediately each time a packet of flow k is dequeued. Should $V(t)$ and $V^k(t)$ have been computed exactly, then, by definition of $V^k(t)$, $W^k(t_1,t_2) = \phi^k V^k(t_1,t_2)$ would have held. In contrast, the fact that $V^k(t)$ immediately increments also implies that $W^k(t_1,t_2) \ge \phi^k V^k(t_1,t_2) - L^k$ holds. Substituting (24) in the latter inequality, we get

$$W^k(t_1,t_2) \ge \phi^k W(t_1,t_2) - 2\phi^k L - 3\phi^k\sigma_i - L^k. \tag{25}$$

The other case is when flow k is not eligible at time t_1. This implies that $V^k(t_1)$ is exactly equal to $S^k(t_1)$. Hence, considering that $S^k(t_1) \le V(t_1) + \sigma_i$ and exploiting the same properties used in the derivations for the other case, we have

$$\begin{aligned} W^k(t_1,t_2) &\ge \phi^k(S^k(t_2) - S^k(t_1)) - L^k \\ &\ge \phi^k(S_i(t_2) - S^k(t_1)) - L^k \\ &> \phi^k(V(t_2) - L - 2\sigma_i - (V(t_1) + \sigma_i) - L^k \\ &> \phi^k W(t_1,t_2) - 2\phi^k L - 3\phi^k\sigma_i - L^k. \end{aligned} \tag{26}$$

■

Comparison to Other Schedulers: In a perfectly fair ideal fluid system such as the GPS server, $\text{B-WFI}^k = 0$ (see [2]), whereas repeating the same passages of the proof in case of exact timestamps (i.e., exact $\text{WF}^2\text{Q+}$ *with stepwise* $V(t)$), the resulting B-WFI^k would be $L^k + 2\phi^k L$.

The B-WFIs for S-KPS and GFQ have not been computed by their authors. However, both these algorithms and QFQ implement the same policy ($\text{WF}^2\text{Q+}$), differing only in how they approximate the timestamps. Generalizing the previous proof, we can show that GFQ has a slightly lower $\text{B-WFI}^k_{\text{GFQ}} =$

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

$2\phi^k\sigma_i + 2\phi^k L + L^k$, whereas S-KPS has $\text{B-WFI}^k_{S-KPS} = \text{B-WFI}^k_{QFQ} = 3\phi^k\sigma_i + 2\phi^k L + L^k$.

Relative Fairness: The relative fairness bound, RFB, is defined as the maximum difference, over any time interval $[t_1, t_2]$ and pair of flows k and p, between the normalized service given to two continuosly backlogged flows

$$\text{RFB} \equiv \max_{\forall k,p,[t_1,t_2]} \left| \frac{W^k(t_1,t_2)}{\phi^k} - \frac{W^p(t_1,t_2)}{\phi^p} \right|. \quad (27)$$

Consider two flows, k and p, belonging, respectively, to groups i and j, and continuously backlogged during a time interval $[t_1, t_2]$. Equation (6), Theorem 2, and the fact that a group is served only if eligible, give an upper bound to the normalized service received by a flow in the interval, resulting in $\frac{W^k(t_1,t_2)}{\phi^k} \le W(t_1, t_2) + L + 4\sigma_i$. The proof of Theorem 7 establishes a lower bound for the normalized service. Substituting these two extremes in (27) and taking the maximum over all possible flow/group pairs, we have

$$\text{RFB} \le 3L + \max\left(4\sigma_i + 3\sigma_j + \frac{L^p}{\phi^p}, 4\sigma_j + 3\sigma_i + \frac{L^k}{\phi^k}\right). \quad (28)$$

As for GFQ and S-KPS, we have $\text{RFB}_{\text{S-KPS}} = 2L + \frac{L^k}{\phi^k} + \frac{L^p}{\phi^p} + 3(\sigma_i + \sigma_j)$ and $\text{RFB}_{\text{GFQ}} = 2L + \frac{L^k}{\phi^k} + \frac{L^p}{\phi^p} + 2(\sigma_i + \sigma_j)$. To put the bound for FRR in a form that allows it to be compared more easily against the bound for QFQ, we assume that all packets have the same length (otherwise RFB_{FRR} may be higher), and get $\text{RFB}_{\text{FRR}} = \max(4\sigma_i + \frac{L}{\phi^k} + 10\sigma_j, 4\sigma_j + \frac{L}{\phi^p} + 10\sigma_i)$ (in the best case for FRR, i.e., for $C = 2$).

B. Time Guarantees

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

$$\text{T-WFI}^k \equiv \max\left(t_c - t_a - \frac{Q^k(t_a^+)}{\phi^k R}\right) \quad (29)$$

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

Theorem 8: T-WFI for QFQ. For a flow k belonging to group i, QFQ guarantees

$$\text{T-WFI}^k = (3\sigma_i + 2L)\frac{1}{R}. \quad (30)$$

The proof, omitted for brevity, is conceptually similar to the one for the B-WFI. Here again, note that the factor σ_i/R (or equivalent) is present in the T-WFI of any near-optimal scheduler. For comparison, a perfectly fair ideal fluid system would have $\text{T-WFI}^k = 0$, whereas for $\text{WF}^2\text{Q+}$, which uses exact timestamps, repeating the same passages of the proof yields $\text{T-WFI}^k = (\frac{L^k}{\phi^k} + 2L)/R$.

Same as for the B-WFI, the T-WFI of S-KPS happens to be equal to that of QFQ, whereas the T-WFI of GFQ is lower than that of QFQ by σ_i/R. Finally, about FRR, as already shown in Section II, the T-WFI of FRR in a realistic scenario is not lower than $(24\sigma_i + 19L)\frac{1}{R}$.

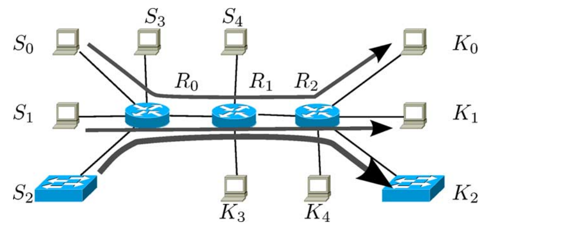

Fig. 7. Simulated scenario. Flows f_0 and f_1 are originated by nodes S_0 and S_1, whereas S_2 generates 50 CBR flows to perturbate the traffic. All routers run the same scheduling algorithm.

VII. Experimental Results

We evaluate the performance of QFQ by comparing its service properties and actual run times to those of other scheduling algorithms. Due to space limitations, we only report a subset of our experimental results. The algorithms selected for the experiments presented here are DRR, to represent the class of high-performance round-robin schedulers; $\text{WF}^2\text{Q+}$, as a reference point for its optimal service properties; and S-KPS, as an example of high-efficiency timestamp-based scheduler.

The experiments cover two aspects: *service properties* are evaluated by running experiments with NS on a simulated topology and measuring end-to-end delays and their variations; *absolute performance* is evaluated by measuring actual run times of production-quality code, i.e., code that includes all features needed in an actual deployment, such as support for dynamic flow creation and destruction, and exception handling. These features are normally neglected in prototype implementations, but are necessary in a realistic test as their support may impose significant overhead to the run times.

A. Evaluation of Service Properties

To prove the effectiveness of the service properties guaranteed by QFQ, we implemented it in the ns2 simulator [1], and we compared it to DRR, S-KPS, and $\text{WF}^2\text{Q+}$.

The network topology used in the simulations is inspired by the one used in [7] and is depicted in Fig. 7. Links R_0-R_1 and R_1-R_2 have 10 Mb/s bandwidth and 10 ms propagation delay; all the other links have 100 Mb/s and 1 ms. The observed flows are f_0 (a 32-kb/s CBR from S_0 to K_0), and f_1 (a 512-kb/s CBR from S_1 to K_1). Interfering flows are a 512-kb/s CBR from S_1 to K_1 (same as f_1), 50 160-kb/s CBR flows from S_2 to K_2, and two best-effort flows, one from S_3 to K_3 and one from S_4 to K_4, each generated from its own Pareto source with mean on and off times of 100 ms, $\alpha = 1.5$, and mean rate of 2 Mb/s (larger than the unallocated bandwidth of the links between the routers, in order to saturate their queues).

Table II shows the end-to-end delays (average, stddev and maximum) experienced by f_0 and f_1 during the last 15 s of simulation (the total simulation time was 20 s; the first five were not considered to let the values settle). QFQ performs as expected, with delays similar to the ones measured for S-KPS, given the common nature that the two schedulers share. DRR shows much larger delays and deviations, which is also expected because of the inherent $O(N)$ WFI of this family of schedulers.

The "max" value for the low bandwidth flows is in good accordance with the WFI values computed in Section VI: The

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

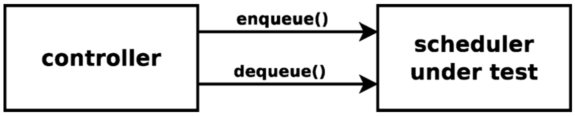

Fig. 8. Our testing environment: A controller drives the scheduler module with programmable sequences of requests.

TABLE II
SIMULATION RESULTS FOR THE TOPOLOGY OF FIG. 7. END-TO-END DELAYS (AVERAGE/STDDEV, MAX) IN MILLISECONDS FOR THE OBSERVED FLOWS

	f_0 (32 Kbit/s) avg $\pm$ stddev	max	f_1 (512 Kbit/s) avg $\pm$ stddev	max
DRR	134.87 ± 34.28	216.99	126.32 ± 30.64	206.40
S-KPS	46.28 ± 5.42	59.91	22.59 ± 0.60	23.34
QFQ	43.16 ± 5.60	47.56	22.76 ± 0.61	23.33
WF^{2}Q+	34.39 ± 0.35	35.20	22.59 ± 0.61	23.33

delay component inversely proportional to the flow's rate is best for WF^{2}Q$+$ and grows as we move to QFQ, S-KPS, and DRR. The larger standard deviation of the delays in QFQ and S-KPS, compared to WF^{2}Q$+$ comes from the use of approximate timestamps, which gives QFQ and S-KPS a WFI larger than that of WF^{2}Q$+$. Also note how the effect of approximations is higher for f_0 (a low rate flow) than for f_1, which is a high rate flow.

B. Run-Time Performance

Together with the good service guarantees, the most interesting feature of QFQ is the constant (independent of the number of flows) and small per-packet execution time, which makes the algorithm extremely practical.

To study the actual performance of our algorithm, and compare it to other alternatives, we have measured the C versions of QFQ and various other schedulers, including S-KPS, which we implemented as part of the Dummynet [4] traffic shaper, running on FreeBSD, Linux, and Windows.

We have performed a thorough performance analysis by running the schedulers in the environment shown in Fig. 8, where we could precisely control the sequence of enqueue/dequeue requests presented to the schedulers. The controller lets us decide the number and distribution of flow weights and packet sizes, as well as keep track of the number of backlogged flows and the total amount of traffic queued in the scheduler. These parameters may impact the behavior of the schedulers, by influencing the code paths taken by the algorithms, and the memory usage and access patterns. The latter are extremely important on modern CPUs, where cached and noncached access times differ by one order of magnitude or more.

In the next section, we report experimental results for the average *enqueue()+dequeue()* times (including generation and disposal by the controller) in different operating conditions (number and distribution of flows, queue size occupation). One of the configurations (the "NONE" case) only measures the controller's costs, so we can determine, by difference, the time consumed by the scheduler.

This test setup does not allow us to separate the cost of *enqueue()* and *dequeue()* operations, but the problem is not relevant. First, in the steady state, there is approximately the same number of calls for the two functions. Only when a link is severely overloaded, the number of *enqueue()* will be much larger than its counterpart, but in this case dropping a packet (in the *enqueue()*) is very inexpensive. Second, in most algorithms it is possible to move some operations between *enqueue()* and *dequeue()*, so it is really the sum of the two costs that counts to judge the overall performance of an algorithm.

Test Cases: Our tests include the following algorithms.

NONE	The baseline case, measures the cost of packet generation and disposal, including memory-touching operations. Packets generated by the controller are stored in a FIFO queue and extracted when the controller calls *dequeue()*.
FIFO	The simplest possible scheduler, an unbounded FIFO queue. Compared to the baseline case, here we exercise the scheduler's API, which causes one extra function calls and counter updates on each request.
DRR	The Deficit Round Robin scheduler, where each flow has a configurable quantum size.
QFQ	QFQ, as described in this paper. We use 19 groups, packet sizes up to 2 kB, and weights between 1 and 2^{16}.
S-KPS	Our implementation from the description in [9], with some minor optimizations, and revised by the original authors. Internal parameters (e.g., $l_{\min}, l_{\max}$) have been set to values similar to those used for QFQ.
WF2Q+	The WF^{2}Q$+$ algorithm taken from the FreeBSD's dummynet code. It has $O(\log N)$ scaling properties, but it is of interest to determine the breakeven point between schedulers with different asymptotical behavior.

Flow Distributions: We ran extensive tests with different combinations and numbers of flows (from 1 to 128 K), with various weight and packet size distributions. These configurations show how the schedulers depend on the number of flows, traffic classes, and also their sensitivity to memory access times.

Load Conditions: To emulate different load conditions for the link, we generate requests for the scheduler with three patterns: *SMALL* and *LARGE* generate bursts of $5N$ and $30N$packets, respectively (where N is the number of active flows), and then completely drain the scheduler; *FULL* keeps the scheduler constantly busy, with a total backlog between $3N$ and $30N$ packets. The bursty patterns try to reproduce operation on a normally unloaded link, whereas the *FULL* pattern mimics the behavior of a fully loaded link driven by TCP or otherwise adaptive flows, which modify their offered load depending on available bandwidth.

C. Results

Table III and Fig. 9 report some of the most significant test results, measured on a low-end desktop machine (2.1 GHz CPU, 32-bit OS, 667 MHz memory bandwidth), with code compiled with gcc -O3. Different platforms perform proportionally to the platform's performance (e.g., a 3-GHz Nehalem CPU is almost twice as fast; a 200-MHz MIPS CPU on a low-cost Access Point

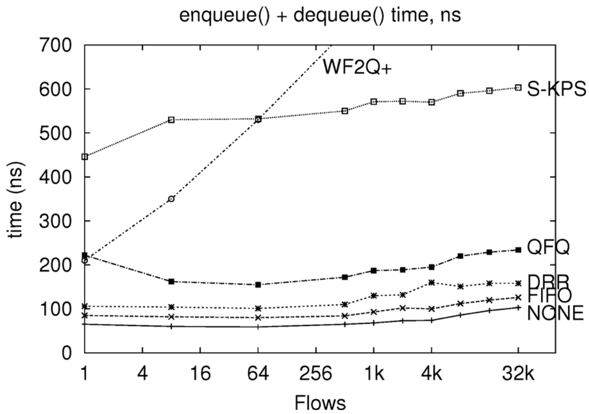

Fig. 9. Scaling properties of the various algorithms. $\mathrm{WF}^2\mathrm{Q}+$ grows as $O(\log N)$, reaching 2000 ns for 32 K flows (see Table III).

TABLE III
SUBSET OF EXPERIMENTAL RESULTS, FOR DIFFERENT FLOW DISTRIBUTIONS AND LOAD CONDITIONS

Time (nanoseconds) for an *enqueue()/dequeue()* pair and packet generation. **Standard deviations are within 3% of the average** (not reported to reduce the clutter in the table)

8 flows						
	NONE	FIFO	DRR	**QFQ**	S-KPS	$\mathrm{WF}^2\mathrm{Q}+$
SMALL	62	80	102	**168**	458	356
LARGE	60	82	104	**162**	530	350
FULL	60	80	102	**163**	543	344
512 flows						
	NONE	FIFO	DRR	**QFQ**	S-KPS	$\mathrm{WF}^2\mathrm{Q}+$
SMALL	62	82	111	**170**	468	732
LARGE	65	84	110	**172**	550	730
FULL	64	85	110	**175**	560	740
32768 flows						
	NONE	FIFO	DRR	**QFQ**	S-KPS	$\mathrm{WF}^2\mathrm{Q}+$
SMALL	90	114	185	**230**	550	1900
LARGE	103	126	158	**234**	603	1880
FULL	62	117	147	**222**	601	1690
1:32k,2:4k,4:2k,8:1k,128:16,1k:1 flows						
	NONE	FIFO	DRR	**QFQ**	S-KPS	$\mathrm{WF}^2\mathrm{Q}+$
SMALL	91	120	167	**247**	598	1868
LARGE	107	131	160	**250**	595	1734
FULL	92	119	160	**255**	612	1715

is 30–40 times slower in running the same experiments). Similarly, the point where cache effects become visible varies depending on available cache sizes.

Fig. 9 shows clearly that the performance of all $O(1)$ algorithms does not depend on the number of flows (except for the impact of cache misses), whereas WF2Q+ shows the expected $O(\log N)$ behavior. We see that DRR and FIFO are extremely fast, with most of the time in the test consumed by the packet generator (the curve labeled NONE in the figure), which accounts for approximately 60 ns per enqueue/dequeue pair. All schedulers, and the generator itself, show a modest increase of the execution time as the number of flows goes (on this particular platform) above 4 K. This is likely due to the working set of the algorithm overflowing the available cache, which causes cache misses that impact on the total execution time. In absolute terms, QFQ behaves really well, consuming about 100–110 ns (excluding the traffic generation) up to the point where cache misses start to matter. S-KPS also has reasonably good performance, taking approximately 500 ns ($2.5 \ldots 3$ times the cost of QFQ).

Finally, we would like to note that while $\mathrm{WF}^2\mathrm{Q}+$ has obvious scalability issues, it can still be a viable alternative for configurations with a small number of flows.

The final block of the table reports the result of experiments with a large mix of flows using different weights. This case does not show significant differences with the case where all flows have the same parameters.

Table III, Fig. 9, and other experiments not reported here show that algorithms can have peculiar behaviors in certain conditions. As an example, QFQ takes a modest performance hit when there is only one flow backlogged. This happens because, in the dequeue code, the removal of the flow from the group leaves the group empty and triggers unnecessary calls to *unblock_groups()* and *make_eligible()*. S-KPS seems to have slightly better performance in the presence of small bursts, presumably due to similar reasons (certain code paths becoming more frequent). DRR takes a performance hit when the packet size is not matched with the quantum size, as certain packets require two rounds instead of one to be processed. These variations tend to be small in absolute and relative terms, but are measurable as we are dealing with extremely fast algorithms where even small changes in the instruction counts affect the performance.

QFQ is a significant step toward the feasibility of software packet processing on 10-Gb/s links. At these speeds, the per-packet budget varies between 67.2 and 1230 ns per packet (for 64- and 1518-B frames). QFQ's speed (100–150 ns/pkt) fits well in the budget, together with recent results on fast packet I/O [14].

VIII. CONCLUSION

In this paper, we presented QFQ, an approximate implementation of $\mathrm{WF}^2\mathrm{Q}+$ that can run in true constant time, with very low constants and using extremely simple data structures. The algorithm is based on very simple instructions and uses very small and localized data structures, which make it amenable to a hardware implementation. Together with a detailed description of the algorithm, we provide a theoretical analysis of its service properties and present an accurate performance analysis, comparing QFQ to a variety of other schedulers. The experimental results show that QFQ lives up to its promises: It is faster than other schedulers with optimal service guarantees, only two times slower than DRR, and operates, even in software, at a rate compatible with 10-Gb/s interfaces.

QFQ and the other algorithms analyzed here are available at [5] as well as part of standard distributions of FreeBSD and Linux and are included in the Dummynet [4] traffic shaper/network emulator for FreeBSD, Linux, and Windows.

REFERENCES

[1] "The network simulator—ns-2," 2012 [Online]. Available: http://www.isi.edu/nsnam/ns/
[2] J. C. R. Bennet and H. Zhang, "Hierarchical packet fair queueing algorithms," *IEEE/ACM Trans. Netw.*, vol. 5, no. 5, pp. 675–689, Oct. 1997.

[3] J. C. R. Bennett and H. Zhang, "$\mathrm{WF}^2\mathrm{Q}$: Worst-case fair weighted fair queueing," in *Proc. IEEE INFOCOM*, Mar. 1996, pp. 120–128.
[4] M. Carbone and L. Rizzo, "Dummynet revisited," *Comput. Commun. Rev.*, vol. 40, no. 2, pp. 12–20, 2010.
[5] F. Checconi, P. Valente, and L. Rizzo, "QFQ: Efficient packet scheduling with tight bandwidth distribution guarantees (full paper and experimental code)," 2012 [Online]. Available: http://info.iet.unipi.it/~luigi/qfq/
[6] C. Guo, "SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks," in *Proc. ACM SIGCOMM*, Aug. 2001, pp. 211–222.
[7] C. Guo, "G-3: An $O(1)$ time complexity packet scheduler that provides bounded end-to-end delay," in *Proc. IEEE INFOCOMM*, May 2007, pp. 1109–1117.
[8] M. Karsten, "SI-$\mathrm{WF}^2\mathrm{Q}$: $\mathrm{WF}^2\mathrm{Q}$ approximation with small constant execution overhead," in *Proc. IEEE INFOCOM*, Apr. 2006, pp. 1–12.
[9] M. Karsten, "Approximation of generalized processor sharing with stratified interleaved timer wheels," *IEEE/ACM Trans. Netw.*, vol. 18, no. 3, pp. 708–721, Jun. 2010.
[10] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts, "Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing," *Perform. Eval. Rev.*, vol. 33, no. 1, pp. 217–228, 2005.
[11] L. Lenzini, E. Mingozzi, and G. Stea, "Tradeoffs between low complexity, low latency, and fairness with deficit round-robin schedulers," *IEEE/ACM Trans. Netw.*, vol. 12, no. 4, pp. 681–693, Aug. 2004.
[12] L. L. Peterson and B. S. Davie, *Computer Networks—A Systems Approach*. San Mateo, CA: Morgan Kaufmann, 2010.
[13] S. Ramabhadran and J. Pasquale, "The stratified round robin scheduler: Design, analysis and implementation," *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 1362–1373, Dec. 2006.
[14] L. Rizzo, "Revisiting network I/O APIs: The netmap framework," *Commun. ACM*, vol. 55, no. 3, pp. 45–51, Mar. 2012.
[15] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round robin," *IEEE/ACM Trans. Netw.*, vol. 4, no. 3, pp. 375–385, Jun. 1996.
[16] S. J. Golestani, "A self-clocked fair queueing scheme for broadband applications," in *Proc. IEEE INFOCOM*, Jun. 1994, pp. 636–646.
[17] D. C. Stephens, J. C. R. Bennett, and H. Zhang, "Implementing scheduling algorithms in high-speed networks," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1145–1158, Jun. 1999.
[18] D. Stiliadis and A. Varma, "A general methodology for designing efficient traffic scheduling and shaping algorithms," in *Proc. IEEE INFOCOM*, Apr. 1997, pp. 326–335.
[19] I. Stoica and H. Abdel-Wahab, "Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation," Department of Computer Science, Old Dominion University, Norfolk, VA, Tech. Rep. 95-22, Nov. 1995.
[20] S. Suri, G. Varghese, and G. Chandranmenon, "Leap forward virtual clock: A new fair queuing scheme with guaranteed delays and throughput fairness," in *Proc. IEEE INFOCOM*, Apr. 1997, pp. 557–565.
[21] P. Valente, "Exact GPS simulation and optimal fair scheduling with logarithmic complexity," *IEEE/ACM Trans. Netw.*, vol. 15, no. 6, pp. 1454–1466, Dec. 2007.
[22] P. Valente, "Extending $\mathrm{WF}^2\mathrm{Q}+$ to support a dynamic traffic mix," in *Proc. AAA-IDEA*, Jun. 2005, pp. 26–33.
[23] J. Xu and R. J. Lipton, "On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms," *IEEE/ACM Trans. Netw.*, vol. 13, no. 1, pp. 15–28, Feb. 2005.
[24] X. Yuan and Z. Duan, "Fair round-robin: A low complexity packet scheduler with proportional and worst-case fairness," *IEEE Trans. Comput.*, vol. 58, no. 3, pp. 365–379, Mar. 2009.

Fabio Checconi received the Ph.D. degree in computer engineering, with a focus on real-time operating systems, from the Scuola Superiore Sant'Anna, Pisa, Italy, in 2010.

He currently is a Postdoctoral Researcher with the IBM T. J. Watson Research Center, Yorktown Heights, NY, where he works in the Multicore Systems Group.

Luigi Rizzo received the Laurea degree from the Università di Pisa, Pisa, Italy, in 1987, and the Ph.D. degree from the Scuola Superiore Sant'Anna, Pisa, Italy, in 1993, both in electronic engineering.

He is an Associate Professor in computer networks with the Università di Pisa. He Rizzo has done extensive work in the areas of computer networks, network emulation, and operating systems.

Paolo Valente received the Laurea and Ph.D. degrees in computer systems engineering from the University of Pisa, Pisa, Italy, in 2000 and 2004, respectively.

Since 2006, he has been an Assistant Professor (Ricercatore) with the Department of Computer Science, University of Modena, Modena, Italy. His current research focus is the design and analysis of CPU, network, and disk scheduling algorithms.